

Community Experience Distilled

Django Design Patterns and Best Practices

Easily build maintainable websites with powerful and relevant Django design patterns

Arun Ravindran

[PACKT] open source
PUBLISHING

目錄

简介

所有译文同时以 **GitHub Issue** 的形式发布，点此阅读。

版权协议

Django-Design-Patterns-and-Best-Practices

中文名：《Django 设计模式与最佳实践》
英文原版：https://www.packtpub.com/web-development/django-design-patterns-and-best-practices
作者：Arun Ravindran
出版日期：March 2015
特色：Easily build maintainable websites with powerful and relevant Django design patterns
级别：Mastering
页数：Paperback 222 pages

第三章预览

第三章 模型

本章，我们会讨论以下话题：

- 模型的重要性
- 类图表
- 模型的结构模式

- 模型的行为模式
- 迁移

M大于V与C

在Django中，模型就是类，该类提供了一种处理数据库的面向对象方法。通常，每个类都引用一个数据库表，每个属性都引用一个数据库列。你可以使用一个自动生成的API来查询这些表。

模型是很多其他组件的基础。只要你有一个模型，你可以很快地得到模型admin，模型表单，以及所有类型的通用视图。每种情况下，都需要你编写一两行代码，这样可以让它看上去没有太多魔法。

模型也被用在更多的超出你期望的地方。这是因为Django可以以多种方式运行。Django的一些切入点如下：

- 常见的web请求-响应流程
- Django的交互式命令行
- 管理命令
- 测试脚本
- 异步任务队列，比如Celery

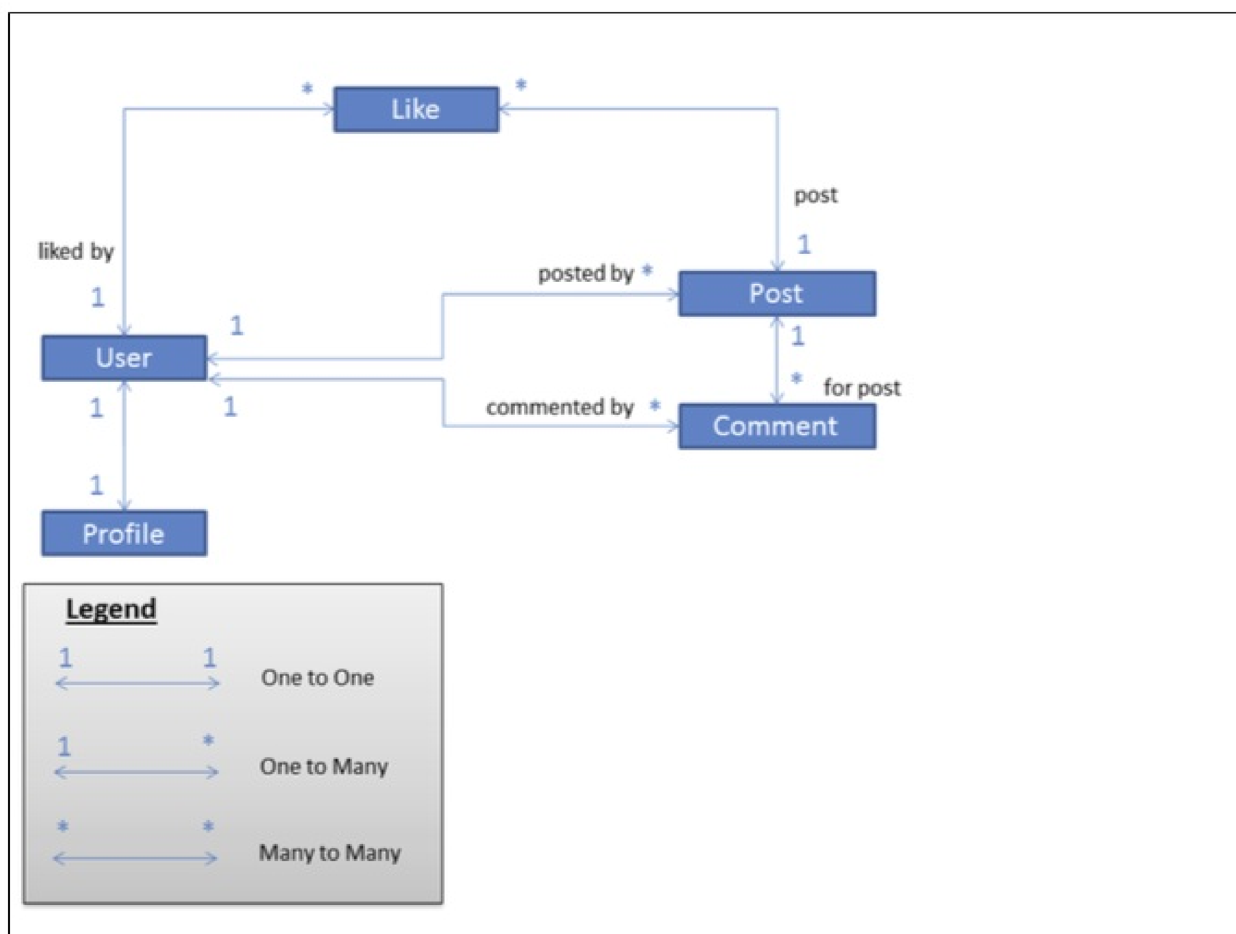
几乎所有的情况中，模型模块都需要导入（作为django.setup()的一部分）。因此，最好保证模型远离任何不必要的依赖，或者导入任何的其他Django组件，比如视图。

简而言之，恰当地设计模型是件十分重要的事情。现在，让我们从SuperBook模型设计开始。

> 注释
>
> 自带午餐便当
> *作者注释：SuperBook项目的进度会以这样的一个盒子来表现。你可以跳过这个盒子，但是在web应用项目中的情况下，你缺少的是领悟，经验 。
>
> 史蒂夫和客户的第一周——超级英雄情报监控（简称为S.H.I.M）。简单来说，这是一个大杂烩。办公室是非常未来化的，但是不论做什么事情都需要上百个审核和签字。
>
> 作为Django开发者的领队，史蒂夫已经配置好了中型的运行超过两天的4台虚拟机。第二天的一个早晨，机器自己不翼而飞了。一个附近的清洁机器人说，机器被法务部们给带走了，他们要对未经审核的软件安装做出处理。
>
> 然而，CTO哈特给予史蒂夫了极大的帮助。他要求机器在一个小时之内完好无损地给还回去。他还对SuperBook项目做出了提前审核以避免将来可能出现的任何阻碍。
>
> 那个下午的稍晚些时候，史蒂夫给他带了一个午餐便当。身着一件米色外套和浅蓝色牛仔裤的哈特如约而至。尽管高出周围人许多，有着清爽面庞的他依旧那么帅气，那么平易近人。他问史蒂夫如果他之前是否尝试过构建一个60年代的超级英雄数据库。
>
> ”嗯，对的，是哨兵项目么？“，史蒂夫说道。”是我设计的。数据库看上去被设计成了一个条目-属性-值形式的模式，有些地方我考虑用反模式。可能，这些天他们有一些超级英雄属性的小想法。哈特几乎等不到听完最后一句，他压低嗓门道：“没错。是我的错。另外，他们只给了我两天来设计整个架构。他们这是在要我老命啊！”
>
> 听了这些，史蒂夫的嘴巴张的大大的，三明治也卡在了嘴里。哈特微笑着道：“当然了，我还没有尽全力来做这件事。只要它成长为100万美元的单子，我们就可以多花点时间在这该死的数据库上了。SuperBook用它就能分分钟完事的，小史你说呢？”
>
> 史蒂夫微微点头称是。他从来没有想过在这么样的地方将会有上百万的超级英雄出现。

模型搜寻

这是我们头一次见识到SuperBook中模型。我们只表示了基本模型，以及类图表中的表单的基本关系，这也是早期尝试中所特有的情况：



让我们暂且忘掉模型，来谈谈我们正在构建的对象的术语。每个用户都有一个账户。用户可以写多个回复或者多篇文章。**Like**同时关联到了一个独立用户/文章组合。

建议你为自己的模型画一个这样类图表。这一步的某些属性缺失了，不过你可以在之后对它们进行详细补充。只要整个项目用图表表现出来，便可以轻松地分离app了。

下面是创建这个表现的一些提示：

- 盒子表示条目，它将成为模型。
- 名词通常作为条目的终止。
- 箭头是双向的，它代表了Django中的三种关系类型其中的一种：一对一，一对多（通过外键实现），和多对多。
- 字段表明在模型中根据条目-关系模型（**ER-modle**）定义了一对多关系。换句话说，星号就是声明外键的地方。

类图表可以映射到下面的Django代码中（分布于多个应用之中）：

```
class Profile(models.Model):
    user = models.OnToOneField(User)

class Post(models.Model):
    posted_by = models.ForeignKey(User)

class Comment(models.Model):
    commented_by = models.ForeignKey(User)
    for_post = models.ForeignKey(Post)

class Like(models.Model):
    liked_by = models.ForeignKey(User)
    post = models.ForeignKey(Post)
```

后面，我们不会直接地引用**User**，而是使用更常见的**settings.AUTH_USER_MODEL**来。

把model.py分到多个文件中去

就像多数的Django组件那样，一个大的model.py文件可以在一个包内分割为多个文件。**package**通过一个目录来实现，它包含多个文件，目录中的一个文件必须是一个称为 `__init__.py` 特殊文件。

所有可以在包级别中暴露的定义都必须在 `__init__.py` 里使用全局变量域定义。例如，如果我们分割model.py到独立的类，models子文件夹中的对应文件，比如，postable.py，post.py和comment.py, 之后 `__init__.py` 包会像这样：

```
    from postable import Postable
    from post import Post
    from commnet import Comment
```

现在你可以像之前那样导入models.Post了。

在 `__init__.py` 包中的任何其他代码都会在包运行时被导入。因此，它是一个任意级别包初始化代码的理想之地。

结构模式

本节包含多个帮助你设计和构建模型的设计模式。

模式-规范化模型

问题：通过设计，模型实例的重复数据引起数据不一致。

解决方法：通过规范化，分解模型到更小的模型。使用这些模型之间的逻辑关系来连接他们。

问题细节

想象一下，如果某人用下面的方法设计Post表（省略部分列）：

超级英雄的名字	消息	发布时间
Captain Temper	消息已经发布过了？	2012/07/07/07:15
Professor English	应该用“Is”而不是“Has"	2012/07/07/07:17
Captain Temper	消息已经发布过了？	2012/07/07/07:18
Capt. Temper	消息已经发布过了？	2012/07/07/07:19

我希望你注意到了在最后一行的超级英雄名字和之前不一致（船长一如既往的缺乏耐心）。

如果我们看看第一列，我们也不确定哪一个拼写是正确的——`Captain Temper或者Capt.Temper`。这就是我们要通过规范化消除的一种数据冗余。

详解

在我们看下完整的规范方案，让我们用Django模型的上下文来个关于数据库规范化的简要说明。

规范化的三个步骤

规范化有助于你更有效地的存储数据库。只要模型完全地的规范化处理，他们就不会有冗余的数据，每个模型应该只包含逻辑上关联到自身的数据。

这里给出一个简单的例子，如果我们规范化了Post表，我们就可以不模棱两可地引用发布消息的超级英雄，然后我们需要用一个独立的表来隔离用户细节。默认，Django已经创建了用户表。因此，你只需要在第一列中引用发布消息的用户的ID，一如下表所示：

用户**ID**	消息	发布时间
12	消息已经发布过了？	2012/07/07/07:15
8	应该用“Is”而不是“Has"	2012/07/07/07:17
12	消息已经发布过了？	2012/07/07/07:18
12	消息已经发布过了？	2012/07/07/07:19

现在，不仅仅相同用户发布三条消息的清楚在列，而且我们可以通过查询用户表找到用户的正确的名字。

通常来说，你会按照模型的完全规规范化表来设计模型，也会因为性能原因而有选择性地非规范化设计。在数据库中，**Normal Forms**是一组可以被应用于表，确保表被规范化的指南。一般我们建立第一，第二，第三规范表，尽管他们可以递增至第五规范表。

这接下来的例子中，我们规范化一个表，创建对应的Django模型。想象下有个名字称做“Sightings”的表格，它列出了某人第一次发现超级英雄使用能力或者特异功能。每个条目都提到了已知的原始身份，超能力，和第一次发现的地点，包括维度和经度。

名字	原始信息	能力	第一次使用记录（维度，经度，国家，时间）
Blitz	Alien	Freeze Flight	+40.75, -73.99; USA; 2014/07/03 23:12
Hexa	Scientist	Telekinesis Flight	+35.68, +139.73; Japan; 2010/02/17 20:15
Traveller	Billonaire	Time travel	+43.62, +1.45, France; 2010/11/10 08:20

上面的地理数据提取自http://www.golombek.com/locations.html.

第一范式表（1NF）

确认一下的第一张范式表格，这张表必须含有：

```
多个没有属性（cell）的值
一个主键作为单独一列或者一组列（合成键）
```

让我们试着把表格转换为一个数据库表。明显地，我们的 `Power` 列破坏了第一个规则。

更新过的表满足第一范式表。主键（用一个 `*` 标记）是 `Name` 和 `Power` 的合并，对于每一排它都应该是唯一的。

Name*	Origin	Power*	Latitude	Longtitude	Country	Tin
Blitz	Alien	Freeze	+40.75170	-73.99420	USA	2014/(23:
Blitz	Alien	Flight	+40.75170	-73.99420	USA	2013/(11:
Hexa	Scientist	Telekinesis	+35.68330	+139.73330	Japan	2010/(20:
Hexa	Scientist	Filght	+35.68330	+139.73330	Japan	2010/(20:
Traveller	Billionaire	Time tavel	+43.61670	+1.45000	France	2010/ 08:

第二范式表

第二范式表必须满足所有第一范式表的条件。此外，它必须满足所有非主键列都必须依赖于整个主键的条件。

我们注意到在前面的表中 `Origin` 只依赖于超级英雄，即，`Name` 。不论我们谈论的是哪一个 `Power` 。因此，`Origin` 不是完全地依赖于合成组件- `Name` 和 `Power` 。

这里，让我们只取出原始信息到一个独立的，称做 `Origins` 的表：

Name*	Origin
Blitz	Alien
Hexa	Scientist
Traveller	Billionaire

现在 `Sightings` 表更新为兼容第二范式表，它大概是这个样子：

Name*	Power*	Latitude	Longtitude	Country	Time	
Blitz	Freeze	+40.75170	-73.99420	USA	2014/07/03 23:12	
Blitz		Flight	+40.75170	-73.99420	USA	201 11::
Hexa	Telekinesis	+35.68330	+139.73330	Japan	2010/02/17 20:15	
Hexa	Filght	+35.68330	+139.73330	Japan	2010/02/19 20:30	
Traveller	Time tavel	+43.61670	+1.45000	France	2010/11/10 08:20	

第三范式表

在第三范式表中，比表格必须满足第二范式表，而且应该额外满足所有的非主键列都直接依赖整个主键，而且这些非主键列都是互相独立的这个条件。

考虑下 `Country` 类。给出 `维度` 和 `经度` ，你可以轻松地得出 `Country` 列。即使观测到超级英雄的地方依赖于 `Name-Power` 合成键，但只是间接地依赖他们。

因此，我们把详细地址分离到一个独立的国家表格中：

Location ID	Latitude*	Longtitude*	Country

1|+40.75170|-73.99420|USA| 2|+35.68330|+139.73330|Japan|
3|+43.61670|+1.45000|France|

现在 `Sightings` 表格的第三范式表大抵如此：

	User ID*	Power*	Location ID	Time
2	Freeze	1	2014/0703 23:12	
2	Flight	1	2013/03/12 11:30	
4	Telekinesis	2	2010/02/17 20:15	
4	Flight	2	2010/02/19 20:30	
7	Time tavel	3	2010/11/10 08:20	

如之前所做的那样，我们用对应的`User ID`替换了超级英雄的名字，这个用户ID用来引用用户表格。

Django模型

现在我们可以看看这些规范化的表格可以用来表现Django模型。Django中并不直接支持合成键。这里用到的解决方案是应用代理键，以及在`Meta`类中指定`unique_together`属性：

```
class Origin(models.Model):
    superhero = models.ForeignKey(settings.AUTH_USER_MODEL)
    origin = models.CharField(max_length=100)

class Location(models.Model):
    latitude = models.FloatField()
    longtitude = models.FloatField()
    country = models.CharField(max_length=100)

    class Meta:
        unique_together = ("latitude", "longtitude")

class Sighting(models.Model):
    superhero = models.ForeignKey(settings.AUTH_USER_MODEL)
    power = models.CharField(max_length=100)
    location = models.ForeignKey(Location)
    sighted_on = models.DateTimeField()

    class Meta:
        unique_together = ("superhero", "power")
```

性能和非规范化

规范化可能对性能有不利的影响。随着模型的增长，需要应答查询的连接数也随之增加。例如，要在美国发现具有冷冻能力的超级英雄的数量，你需要连接四个表格。先前的内容规范后，任何信息都可以通过查询一个单独的表格被找到。

你应该设计模式以保持数据规范化。这可以维持数据的完整。然而，如果你面临扩展性问题，你可以有选择性地从这些模型取得数据以生成非规范化的数据。

> 提示
>
> 最佳实践 因设计而规范，又因优化而非规范
>
> 例如，在一个确定的国家中计算观测次数是非常普通的，然后将观测次数作为一个附加的字段到 `Location` 模型。现在，你可以使用Django ORM 继承其他的查询，而不是一个缓存的值。
>
> 然而，你需要在每次添加或者移除观测时更新这个计数。你需要添加该计算到*Singhting* 的 `save` 方法，添加一个信号处理器，甚至使用一个异步任务去计算。
>
> 如果你有一个跨越多个表的负责查询，比如国家的超能力计算，你需要创建一个独立的非规范表格。就像前面那样，我们需要在每一次规范化模型中的数据改变时更新这个非规范的表格。
>
> 令人惊讶的是非规范化在大型的网站中是非常普遍的，因为它是数度和存储空间两者之间的折衷。今天的存储空间已经比较便宜了，然而速度也是用户体验中至关重要的一环。因此，如果你的查询耗时过于久的话，那么就需要考虑非规范化了。

我们应该一直使用规范化吗？

过多的规范化是是件不必要的事。有时候，它可以引入一个非必需的能够重复更新和查询的表格。

例如，你的 `User` 模型或许有好多个家庭地址的字段，你可以规范这些字段到一个 `Address` 模型中。可是，多数情况下，把一个额外的表引进数据库是没有必要的。

与其针对大多数的非规范化设计，不如在代码重构之前仔细地衡量每个非规范化的机会，对性能和速度上做出一个折衷的选择。

模式-模型mixins

问题：明显地模型含有重复的相同字段/或者方法，违反了DRY原则。

方案：提取公共字段和方法到各种不同的可重复使用的模型mixins中。

问题细节

设计模型之时，你或许某些个公共属性或者行为跨类共享。烈日， `Post` 和 `Comment` 模型需要一直跟踪自己的 `created` 日期和 `modified` 日期。手动地复制-粘贴字段和它们所关联的方法并不符合DRY原则。

由于Django的模型是类，像合成以及继承这样的面向对象方法都是可以选择的解决方案。然而，合成（具有包含一个共享类实例的属性）需要一个额外的间接地访问字段的标准。

继承是有技巧的。我们使用一个 `Post` 和 `Comment` 的公共基类。然而，在Django中有三种类型的继承：**concrete**（具体），**abstract**（抽象），和**proxy**（代理）。

具体继承的运行是源于基类，就像你在Python类中通常用到的那样。不过，在Django中，这个基类将被映射到一个独立的表中。每次你访问基本字段时，都需要一个准确的连接。这会带来非常糟糕的性能问题。

代理继承只能添加新的行为到父类。你不能够添加新字段。因此，这种情况下它也不大好用。

最后，我们只有托付于抽象继承了。

详解

抽象基类是用于模型之间共享数据和行为的游戏简洁方案。当你定义一个基类时，它在数据中没有创建任何与之对象的表。反而，这些字段是在派生的非基类中创建的。

访问抽象基类字段不要 `JOIN` 语句。有支配的字段结构表也是不解自明的。对于这些优势，大多数的Django项目都使用抽象基类实现公共字段或者方法。

使用抽象模型是局限的：

```
它们不能够拥有外键或者来自其他模型的多对多字段。
它们不能够被实例化或者保存。
它们不能够直接用在查询中，因为它没有管理器。
```

下面是post和comment类如何使用一个抽象基类初始设计的：

```
    class Postable(models.Model):
        created = models.DateTimeField(auto_now_add=True)
        modified = modified.DateTimeField(auto_now=True)
        message = models.TextField(max_length=500)

        class Meta:
            abstract = True


    class Post(Postable):
        ...


    class Comment(Postable):
        ...
```

要将一个模型转换到抽象基类，你需要在它的内部 `Meta` 类中写上 `abstract = True` 。这里的 `Postable` 是一个抽象基类。可是，它不是那么的可复用。

实际上，如果有一个类含有 `created` 和 `modified` 字段，我们在后面就可以在附近的任何需要时间戳的模型中重复使用这个时间戳功能。

模型mixins

模型mixins是一个可以把抽象基类当作父类来添加的模型。不像其他的语法，比如Java那样，Python支持多种继承。因此，你可以列出一个模型的任意数量的父类。

Mixins应该是互相垂直的而且易于组合的。把一个mixin放进基类的列表，这些mixin应该可以正常运行。这样看来，它们在行为上更类似于合成而非继承。

较小的mixin的会更好。不论何时一个mixin变得臃肿，而且又违反了独立响应原则，就要考虑把它重构到一个更小的类。就让一个mixin一次做好一件吧。

在前面的例子中，模型mixin用于更新 `created` 和 `modified` 的时间可以轻松地分解出来，一如下面代码所示：

```
class TimeStampedModel(models.Model):
    created = modified.TimeStampModel(auto_now_add=True)
    modified = modified.DateTimeField(auto_now=True)

    class Meta:
        abstract = True


class Postable(TimeStampedModel):
    message = models.TextField(max_length=500)
    ...

    class Meta:
        abstract = True


class Post(Postable):
    ...


class Comment(Postable):
    ...
```

我们现在有两个超类了。不过，功能之间都完全地独立。mixin可以分离到自己的模块之内，并且另外的上下文中被复用。

模式-用户账户

问题：每一个网站都存储一组不同的用户账户细节。然而，Django的内建User模型旨在针对认证细节。

方案：用一个一对一关系的用户模型，创建一个用户账户类。

问题细节

Django提供一个开箱即用的相当不错的User模型。你可以在创建超级用户或者登录amdin接口的时候用到它。它含有少量的基本字段，比如全名，用户名，和电子邮件。

然而，大多数的现实世界项目都保留了很多关于用户的信息，比如他们的地址，喜欢的电影，或者它们的超能力。打Django1.5开始，默认的User模型就可以被扩展或者替换掉。不过，官方文档极力推荐只存储认证数据，即便是在定制的用户模型中也是如此（毕竟，用户模型也是所属于 `auth` 这个app的）。

某些项目是需要多种类型的用户的。例如，SuperBook可以被超级英雄和非超级英雄所使用。这里或许会有一些公共字段，以及基于用户类型的不同字段。

详解

官方推荐解决方案是创建一个用户账户模型。它应该和用户模型有一个一对一的关系。其余的全部用户信息都存储于该模型：

```
class Profile(models.Model):
    user = models.OnToOneField(settings.AUTH_USER_MODEL, primary_key=True)
```

这里推荐你明确的将 `primary_key` 赋值为 `True` 以阻止类似PostgreSQL这样的数据库后端中的并发问题。剩下的模型可以包含其他的任何用户详情，比如生日，喜好色彩，等等。

设计账户模型之时，建议所有的账户详情字段都必须是非空的，或者含有一个默认值。凭直觉我们就知道用户在注册时是不可能填写完所有的账户细节的。此外，我们也要确保创建账户实例时，信号处理器没有传递任何初始参数。

信号

理论上，每一次用户模型的创建都必须把对应的用户账户实例创建好。这个操作通常利用信号来完成。例如，我们可以使用下面的信号处理器侦听用户模型的 `post_save` 信号：

```
# signals.py
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.conf import settings
from . import models


@receiver(post_save, sender=settings.AUTH_USER_MODEL)
def create_profile_handler(sender, instance, created, **kwargs):
    if not created:
        return
    # Create the profile object, only if it is newly created
    profile = models.Profile(user=instance)
    profile.save()
```

注意账户模型除了用户实例之外没有传递额外的参数。

之前没有指定初始信号代码的地方。通常，它们在 `models.py` 中（这是不可靠的）导入或者执行。不过，随着Django 1.7的app载入重构，应用初始化代码位置的问题也很好的解决了。

首先，为你的应用创建一个 `__init__.py` 包以引用应用的 `ProfileConfig`：

```
default_app_config = "profile.apps.ProfileConfig"
```

接下来是 `app.py` 中，子类 `ProfileConfig` 的方法，可于 `ready` 方法中配置信号：

```
# app.py
from django.apps import AppConfig


class ProfileConfig(AppConfig):
    name = "profiles"
    verbose_name = "User Profiles"


    def ready(self):
        from . import signals
```

随着信号的配置，对所有的用户来说，访问 `user.profile` 应该都返回一个 `Profile` 对象，即使是最新创建的用户也是如此。

Admin

现在，用户的详情为存在admin内的两个不同地方：普通用户admin页面中的认证细节，在一个独立账户admin页面中的相同用户的补充账户详情。但是这样做非常麻烦。

为了操作方便，账户admin可以通过定义一个自定义的 `UserAdmin` 嵌入到默认的用户admin中：

```
# admin.py
from django.contrib import admin
from .models import Profile
from django.contrib.auth.models import User


class UserProfileInline(admin.StackedInline):
    model = Profile


class UserAdmin(admin.UserAdmin):
    inlines = [UserProfileInline]


admin.site.unregister(User)
admin.site.register(User, UserAdmin)
```

多账户类型

假设在应用中你需要几种类型的用户账户。这里需要有一个字段去跟踪用户使用的是哪一种账户类型。账户数据本身需要存储在独立的模型中，或者存储在一个统一的模型中。

建议使用聚合账户的方法，因为它让改变账户类型而不丢失账户细节，并具有灵活性，减小复杂度。在这个房中中，账户模型包含一个所有账户类型的字段超集。

例如，SuperBook会需要一个 `SuperHero` 类型账户，和一个 `Ordinary` （非超集英雄）账户。它可以用一个独立的统一账户模型实现：

```
class BaseProfile(models.Model):
    USER_TYPES = (
        (0, 'Ordinary'),
        (1, 'SuperHero'),
    )

    user = models.OnToOneField(settings.AUTH_USER_MODEL, primary_key=True)
    user_type = models.IntegerField(max_length=1, null=True,     choices=USER_TYPES)
    bio = models.CharField(max_length=200, blank=True, null=True)

    def __str__(self):
        return "{}:{:.20}".format(self.user, self.bio or "")

    class Meta:
        abstract = True


class SuperHeroProfile(models.Model):
    origin = models.CharField(max_length=100, blank=True, null=True)

    class Meta:
        abstract = True


class OrdinaryProfile(models.Model):
    address = models.CharField(max_length=200, blank=True, null=True)

    class Meta:
        abstract = True


class Profile(SuperHeroProfile, OrdinaryProfile, BaseProfile):
    pass
```

我们组织账户细节到多个抽象基类再到独立的关系中。 `BaseProfile` 类包含所有的不关心用户类型的公共账户细节。它也有一个 `user_type` 字段，它持续追踪用户的激活账户。

`SuperHeroProfile` 类和 `OrdinaryProfile` 类分别包含所有到超级英雄和非超级英雄特定账户细节。最后，所有这些基类的 `profile` 类创建了一个账户细节的超集。

使用该方法时要主要的一些细节：

> 所有属于类的字段或者它抽象基类都必须是非空的，或者有一个默认值。
>
> 这个方法或许会为了每个用户而消耗更多的数据库，但是却带来极大的灵活性。
>
> 账户类型的激活和非激活字段都必须在模型外部是可管理的。
>
> 说到，编辑账户的表必须显示合乎目前激活用户类型的字段。

模式-服务模式

问题：模型会变得庞大而且不可管控。当一个模型不止实现一个功能时，测试和维护也会变得困难。

解决方法：重构出一组相关方法到一个专用的 `Service` 对象中。

问题细节

富模型，瘦视图是一个通常要告诉Django新手的格言。理论上，你的视图不应该包含任何其他表现逻辑。

可是，随着时间的推移，代码段不能够放在任意地点，除非你打算将它们放进模型中。很快，模型会变成一个代码的垃圾场。

下面是模型可以 `Service` 对象的征兆：

```
1. 与扩展能服务交互，例如web服务中，检查一个用户具有资格。
2. 帮助任务不会处理数据库，例如，生成一个短链接，或者针对用户的验证码。
3. 牵涉到一个短命的对象时不会存在数据库状态记录，烈日，创建一个AJAX调用的JSON响应。
4. 对于长时间运行的任务设计到了多实例，比如Celery任务。
```

Django中的模型遵循着激活记录模式。理论上，它们同时封装应用罗即数据库访问。不过，要记得保持应用逻辑最小化。

在测试时，如果我们发现没有对数据库建模的必要，甚至不会用到数据库，那么我们需要考虑把分解模型类。建议这种场合下使用 `Service` 对象。

详解

服务对象是封装一个 `服务` 或者和系统狡猾的普通而老旧的Python对象（POPOs）。它们通常保存在一个独立的称为 `service.py` 或者 `utils.py` 的文件。

例如，像下面这样，检查一个web服务是否作为一个模型方法：

```
    class Profile(models.Model):
        ...

        def is_superhero(self):
            url = "http://api.herocheck.com/?q={0}".format(
                self.user.username
                )
            return webclient.get(url)
```

该方法可以使用一个服务对象来重构：

```
from .services import SuperHeroWebAPI

def is_superhero(self):
    return SuperHeroWebAPI.is_superhero(self.user.username)
```

现在服务对象可以定义在 `services.py` 中了：

```
API_URL = "http://api.herocheck.com/?q={0}"

class SuperHeroWebAPI:
    ...

    @staticmehtod
    def is_hero(username):
        url = API_URL.format(username)
        return webclient.get(url)
```

多数情况下， `Service` 对象的方法是无状态的，即，它们基于函数参数不使用任何的类属性来独自执行动作。因此，最好明确地把它们标记为静态方法（就像我们对 `is_hero` 所做的那样）。

可以考虑把业务逻辑和域名逻辑从模型迁移到服务对象中去。这样，你可以在Django应用的外部很好使用它们。
想象一下，由于业务的原因要依据某些用户的名字把这些要成为超级英雄的用户加进黑名单。我们的服务对象稍微改动以下就可以支持这个功能：

```
class SuperHeroWebAPI:
    ...

@staticmethod
def is_hero(username):
    blacklist = set(["syndrome", "kcka$$", "superfake"])
    ulr = API_URL.format(username)
    return username not in blacklist and webclient.get(url)
```

理论上，服务对象自包含的。这使它们不用建模——即数据库，也可以易于测试。它们也易于复用。

Django中，耗时服务以Celery这样的异步任务队列方式执行。通常， `Service` 对象以Celery任务的方式执行操作。这样的任务可以周期性地运行或者延迟运行。

检索模式

本节包含处理模型属性的访问，或者对模型执行查询。

模式-属性字段

问题：模型有以方法实现的属性。可是，这些属性不应该保存到数据库。

**解决方案：对这样的方法使用属性装饰器。

问题详情

模型字段存储每个实例的属性，比如名，和姓，生日，等等。它们存储于数据库之中。可是，我们也需要访问某些派生的属性，比如一个完整的名字和年龄。

它们可以轻易地计算数据库字段，因此它们不需要单独地存储。在某些情况下，它们可以成为一个检查所提供的年龄，会员积分，和激活状态是否合格的条件语句。

简洁明了的实现这个方法是定义比如类似 `get_age` 这样函数：

```
    class BaseProfile(models.Model):
        birthdate = models.DateField()
        #...

        def get_age(self):
            today = datetime.date.today()
            return (today.year - self.birthdate.year) - int(
                (today.month, today.day) < (self.birthdate.month, self.birthdate.day)
                )
```

调用 `profile.get_age()` 会通过计算调整过的月和日期所在的那个年份的不同来返回用户的年龄。

不过，这样调用 `profile.age` 变得更加可读（和Python范）。

详解

Python类可以使用 `property` 装饰器把函数当作一个属性来使用。这样，Django模型也可以较好地利用它。替换前面那个例子中的函数：

```
@property
def age(self):
```

现在我们可以用 `profile.age` 来访问用户的年龄。注意，函数的名称要尽可能的短。

属性的一个重大缺陷是它对于ORM来说是不可访问的，就像模型的方法那样。你不能够在一个 `Queryset` 对象中使用它。例如，这么做是无效的，`Profile.objects.exlude(age__lt=18)。

它也是一个定义一个属性来隐藏类内部细节的好主意。这也正式地称做得墨忒耳定律。简单地说，定律声明你应该只访问自己的直属成员或者“仅使用一个点号”。

例如，最好是定义一个 `profile.birthyear` 属性，而不是访问 `profile.birthdate.year` 。这样，它有助于你隐藏 `birthdate` 字段的内在结构。

> 提示
>
> 最佳实践
> 遵循得墨忒耳定律，并且访问属性时只使用点号
>
> 该定律的一个不良反应是它导致在模型中有多个包装器属性被创建。这使模型膨胀并让它们变得难以维护。利用定律来改进你的模型API，减少模型间的耦合，在任何地方都是可行的。

缓存特性

每次我们调用一个属性时，就要重新计算函数。如果计算的代价很大，我们就想到了缓存结果。因此，下次访问属性，我们就拿到了缓存的结果。

```
from django.utils.function import cached_property
#...
@cached_property
def full_name(self):
    # 代价高昂的操作，比如，外部服务调用
    return "{0} {1}".format(self.firstname, self.lastname)
```

缓存的值会作为Python实例的一部分而保存。只要实例一直存在，就会得到同样的返回值。

对于一个保护性机制，你或许想要强制执行高昂代价操作以确保过期的值不会返回。如此情境之下，设置一个 `cached=False` 这样的关键字参数能够阻止返回缓存值。

模式-定制模型管理器

问题：某些模型的定义的查询被重复地访问，这彻彻底底的违反了DRY原则。

解决方案：定义自定义的管理器以给常见的查询一个更有意义的名字。

问题细节

每一个Django的模型都有一个默认称做 `objects` 的管理器。调用 `objects.all()` 会返回数据库中的这个模型的所有条目。通常，我们只对所有条目的子集感兴趣。

我们应用多种过滤器以找出所需的条目组。挑选它们的原则常常是我们的核心业务逻辑。例如，我们发现使用下面的代码可以通过public访问文章：

```
public = Posts.objects.filter(privacy="public")
```

这个标准在未来或许会改变。我们或许也想要检查文章是否标记为编辑。这个改变或许如此：

```
public = Posts.objects.filter(privacy=POST_PRIVACY.Public, draft=Flase)
```

可是，这个改变需要使在任何地方都要用到公共文章。这让人非常沮丧。这仅需要一个定义这样常见地查询而无需“自我重复”。

详解

`Querysets` 是一个极其有用的抽象概念。它们仅在需要时进行惰性查询。因此，通过链式方法（一个流畅的界面）构建更长的 `Querysets` 并不影响性能。

事实上，应该更多的过滤会使结果数据集缩减。这样做通常可以减少结果的内存消耗。

模型管理器是一个模型获取自身 `Queryset` 对象的便利接口。换句话来讲，它们有助于你使用Django的ORM访问潜在的数据库。事实上，`QuerySet` 对象上管理器以一个非常简单的包装器实现。

```
>>> Post.objects.filter(posted_by__username="a")
[<Post:a: Hello World>， <Post:a: This is Private!>]

>>> Post.objects.get_queryset().filter(posted_yb__username="a")
[<Post:a: Hello World>, <Post:a: This is Private!>]
```

默认的管理器由Django创建，`objects` 有多种方法返回 `Queryset` ，比如 `all` ，`filter` 或者 `exclude` 。可是，它们生成一个到数据库的低级API。

定制管理器用于创建域名指定，更高级的API。这样不仅更加具有可读性而且通过实现细节减轻影响。因此，你能够在更高级的抽象概念上工作，紧密地模型化到域名。

前面的公共文章例子可以轻松地转换到一个定制的管理器：

```
# managers.py
from django.db.models.query import Queryset


class PostQuerySet(QuerySet):
    def public_posts(self):
        return self.filter(privacy="public")

PostManager = PostQuerySet.as_manager
```

这是一个在Django 1.7中从 `QuerySet` 对象创建定制管理器的捷径。不像前面的其他方法，这个 `PostManager` 对象像默认的 `objects` 管理器一样是链式的。

如下所示，使用我们的定制管理器替换默认的 `objects` 管理器也是可行的：

```
from .managers import PostManager

class Post(Postable):
    ...
    objects = PostManager()
```

这样做，访问`public_posts`相当简单：

```
public = Post.objects.public_posts()
```

因此返回值是一个`QuerySet`，它们可以更进一步过滤：

```
public_apology = Post.objects.public_posts().filter(
    message_startwith = "Sorry"
    )
```

`QuerySets`由多个有趣的属性。在下一章，我们可以看看某些带有合并的`QuerySet`的常见地模式。

Querysets的组合动作

事实上，对于它们的名字，`QuerySets`支持多组操作。为了说明，考虑包含用户对象的两个`QuerySets`：

```
>>> q1 = User.objects.filter(username__in["a", "b", "c"])
[<User:a>, <User:b>, <User:c>]
>>> q2 = User.objects.filter(username__in["c", "d"])
[<User:c>, <User:d>]
```

对于一些组合操作可以执行以下动作：

```
*Union*：合并，移除重复动作。使用`q1`|`q2`获得[`<User: a>, <User: b>, <User: c>, <User: d>`]

*Intersection*：找出公共项。使用`q1`|`q2`获得[`<User: c>]

*Difference*：从第一个组中移除第二个组。该操作并不按逻辑来。改用`q1.exlude(pk__in=q2)获得〔<User: a
```

同样的操作我们也可以用`Q`对象来完成：

```
from django.db.models import Q

# Union
>>> User.objects.filter(Q(username__in["a", "b", "c"]) | Q(username__in=["c", "d"]))
[`<User: a>, <User: b>, <User: c>, <User: d>`]

# Intersection
>>> User.objects.filter(Q(username__in["a", "b", "c"]) & Q(username__in=["c", "d"]))
[<User: c>]

# Difference
>>> User.objects.filter(Q(username__in=["a", "b", "c"]) & ~Q(username__in=["c", "d"])
[<User: a>, <User: b>]
```

注意执行动作所使用 `&`（和）以及 `~`（否定）的不同。`Q` 对象是非常强大的，它可以用来构建非常复杂的查询。

可是，`Set` 虽类似但却不完美。`QuerySets` 不像数学上的集合，那样按照顺序来。因此，在这一方面它们更接近Python的列表数据结构。

链接多个Querysets

目前为止，我们已经合并了属于相同基类的同类型 `QuerySets` 。可是，我们或许需要合并来自不同模型的 `QuestSets` ，并对它们执行操作。

例如，一个用户的活动时间表包含了它们自身所有的按照反向时间顺序所发布的文章和评论。之前合并的 `QuerySets` 方法是不会起作用的。一个较为天真的做法是把它们转换到列表，连接并排列这个列表：

```
>>> recent = list(posts)+list(comments)
>>> sorted(recent, key=lambda e: e.modified, reverse=True)[:3]
[<Post: user: Post1>, <Comment: user: Comment1>, <Post: user: Post0>]
```

不幸的是，这个操作已经对惰性的 `QuerySets` 对象求值了。两个列表的内存使用算在一起可能很大内存开销。另外，转换一个庞大的 `QuerySets` 到列表是很慢很慢的。

一个更好的解决方案是使用迭代器减少内存消耗。如下，使用 `itertools.chain` 方法合并多个 `QuerySets` ：

```
>>> from itertools import chain
>>> recent = chain(posts, comments)
>>> sorted(recent, key=lambda e: e.modified, reverse=True)[:3]
```

只要计算 `QuerySets` ，连接数据的开销都会非常搞。因此，重要的是，尽可能长的仅有的不对 `QuerySets` 求值的操作时间。

> ## 提示
>
> 尽量延长 `QuerySets` 不求值的时间。

迁移

迁移让你改变模型时更有信心。说的Django 1.7，迁移已经是开发流程中基本的易于使用的一部分了。

新的基本流程如下：

```
1. 第一次定义模型类的话，你需要运行：

    python manage.py makemigrations <app_label>

2. 这将在`app/migrations/`文件夹内创建迁移脚本。
在同样的（开发）环境中运行以下命令：

    python manage.py migrate <app_label>

3. 这将对数据库应用模型变更。有时候，遇到的问题有，处理默认值，重命名，等等。

4. 普及迁移脚本到其他的环境。通常，你的版本控制工具，例如，Git，会小心处理这事。当最新的源释出时，新的迁移

5. 在这些环境中运行下面的命令以应用模型的改变：

    python manage.py migarte <app_label>

不论何时要将变更应用到模型，请重复以上1-5步骤。
```

如果你在命令里忽略了app标签，Django会在每一个app中发现未应用的变更并迁移它们。

总结

模型设计要正确地操作很困难。它依然是Django开发的基础。本章，我们学习了使用模型时多个常见模式。每个例子中，我们都见到了建议方案的作用，以及多种折衷方案。

这下一章，我们会用视图和URL配置来验证所遇到的常见设计模式。

© Creative Commons BY-NC-ND 3.0 | 我要订阅 | 我要捐助

目录预览

内容目录

    - 调用super()更简单
    - 必须更明确地相对导入
    - HttpRequest and HttpResponse have str and bytes types
    - 异常语法的改变和提高
    - 重组标准库
    - 新东西
        - 使用Pyvenv和Pip
    - 其他的改变
- 更多内容

第一章 Django与模式

在这一章，我们讨论以下话题：

```
我们为什么选择Django？
Django是工作原理
什么是模式？
常见的模式合集
Django中的模式
```

我们为什么选择Django？

每个web应用都不尽相同，就像一件手工制作的家具一样。你几乎会很少发现大批量的生成能够完美地达到你的需求。即使你从一个基本需求开始，比如一个博客或者一个社交网络，你都需要缓慢地开发，

这就是类似Django或者Rails的web框架非常流行的原因。框架加速了开发，而且它带有很多练好的经过实践的内容。

Python可能比其他流行的编程语言具有更多的web框架。

开箱即用的admin接口，它是Django才有的独一无二的特点，早些时候，特别是在数据记录和测试方面它大有裨益。而Django的开发文档作为一个出色的开源项目早已是备受赞誉。

最后，Django在多个高流量的网站中历经实战的考验。它对于常见的攻击比如跨站脚本和跨站请求攻击有着异常敏锐观察。

尽管，在理论上，可能对于所有类型的网站Django不是最佳选择，你可以是使用Django构建任何类型的网站。例如，要构建一个基于web聊天的实时接口，或许你要使用Tornado，但是web引用剩下的部分你可以仍旧使用Django来完成。对于开发你要学会选择正确的工具。

某些内建的特性，比如admin接口，如果你使用过其他的web框架或许让你听上去感觉有点怪怪的。为了Django的设计，就让我们找出它是如何问世的。

Django的历史

When you look at the Pyramids of Egypt, you would think that such a simple and minimal design must have been quite obvious. In truth, they are products of 4,000 years of architectural evolution. Step Pyramids, the initial (and clunky) design, had six rectangular blocks of decreasing size. It took several iterations of architectural and engineering improvements until the modern, glazing, and long-lasting limestone structures were invented.

当你看到埃及金字塔的历史时，你会很明显地认为它是一个简单、小规模的设计。事实上，埃及人的作品是经历了四千年建筑的进化。当你走进金字塔时就能够发现它的原始设计（非常厚重），拥有六个矩形的阶梯式递减的大石块。

Looking at Django you might get a similar feeling. So, elegantly built, it must have been awlessly conceived. n the contrary, it was the result of rewrites and rapid iterations in one of the most high-pressure environments imaginable—a newsroom!

再回头来看看Django你也会有类似的感觉。因此，如果要简洁的构建它，必须经过无知无畏的设想。

In the fall of 2003, two programmers, Adrian Holovaty and Simon Willison, working at the Lawrence Journal-World newspaper, were working on creating several local news websites in Kansas. These sites, including LJWorld.com, Lawrence.com, and KUsports.com—like most news sites were not just content-driven portals chock- full of text, photos, and videos, but they also constantly tried to serve the needs of the local Lawrence community with applications, such as a local business directory, events calendar, classifieds, and so on.

一个框架的诞生

This, of course, meant lots of work for Simon, Adrian, and later Jacob Kaplan Moss who had joined their team; with very short deadlines, sometimes with only a few hours' notice. Since it was the early days of web development in Python, they had to write web applications mostly from scratch. So, to save precious time, they gradually refactored out the common modules and tools into something called "The CMS."

Eventually, the content management parts were spun off into a separate project called the Ellington CMS, which went on to become a successful commercial CMS product. The rest of "The CMS" was a neat underlying framework that was general enough to be used to build web applications of any kind.

By July 2005, this web development framework was released as Django (pronounced Jang-Oh) under an open source Berkeley Software Distribution (BSD) license. It was named after the legendary jazz guitarist Django Reinhardt. And the rest, as they say, is history.

移除魔法

Due to its humble origins as an internal tool, Django had a lot of Lawrence ournalWorldspecific oddities. To ake Django truly general purpose, an effort dubbed "Removing the Lawrence" was already underway.

However, the ost significant refactoring effort that Django developers had to undertake was called "Removing the Magic." This ambitious project involved cleaning up all the warts Django had accumulated over the years, including a lot of magic (an informal term for implicit features) and replacing them with a more natural and explicit Pythonic code. For example, the model classes used to be imported from a magic module called django.models.*, rather than directly importing them from the models.py odule they were defined in.

At that time, Django had about a hundred thousand lines of code, and it was a significant rewrite of the PI. n May , , these changes, alost the sie of a small book, were integrated into Django's development version trunk and released as Django release .. This was a significant step toward the Django .

Django坚持做得更好

Every year, conferences called DjangoCons are held across the world for Django developers to meet and interact with each other. They have an adorable tradition of giving a semi-humorous keynote on "why Django sucks." This could be a member of the Django community, or someone who works on competing web frameworks or just any notable personality.

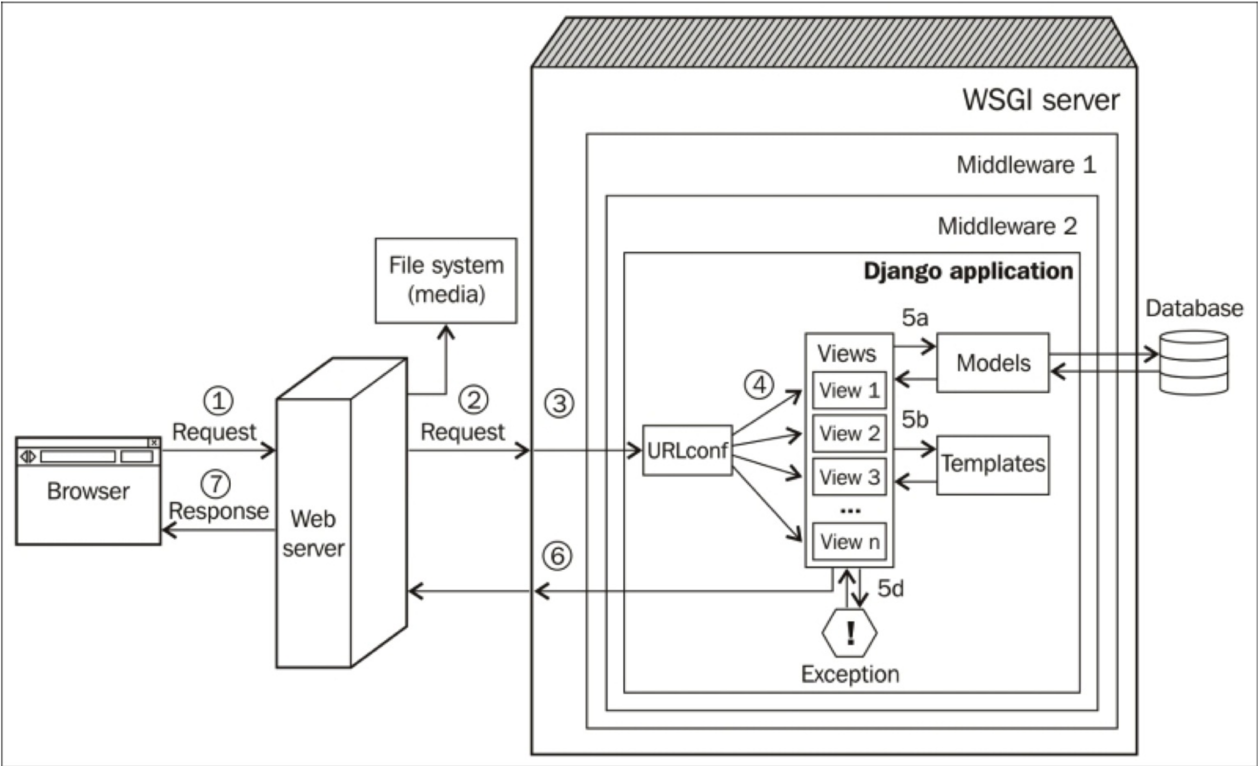
Over the years, it is amazing how Django developers took these criticisms positively and mitigated them in subsequent releases. Here is a short summary of the improvements corresponding to what once used to be a shortcoming in Django and the release they were resolved in:

- New form-handling library (Django 0.96)
- Decoupling admin from models (Django 1.0)
- Multiple database support (Django 1.2)
- Managing static files better Django .
- Better time zone support (Django 1.4)
- Customizable user model (Django 1.5)
- Better transaction handling (Django 1.6)
- Built-in database migrations (Django 1.7)

Django是如何工作的？

要真正的欣赏Django，你需要撇开表象来看本质。它启发你得同时也让会让你不知所措。如果你已经比较了解它，你可以选择跳过这一节。

下图：在Django应用中一个典型的web请求是如何被处理的。



前面的图片展示了从一个访客的浏览器到Django应用并返回的一个web请求的简单历程。如下是数字标识的路径：

```
1. 浏览器发送请求（基本上是字节类型的字符串）到web服务器。

2. web服务器（比如，Nginx）把这个请求转交到一个WSGI（比如，uWSGI），或者直接地文件系统能够取出
一个文件（比如，一个CSS文件）。

3. 不像web服务器那样，WSGI服务器可以直接运行Python应用。请求生成一个被称为environ的Ptyhon字典，
而且，可以选择传递过去几个中间件的层，最终，达到Django应用。

4. URLconf中含有属于应用的urls.py选择一个视图处理基于请求的URL的那个请求，这个请求就已经变成了
HttpRequest——一个Python字典对象。

5. 被选择的那个视图通常要做下面所列出的一件或者更多件事情：

    通过模型与数据库对话。
    使用模板渲染HTML或者任何格式化过的响应。
    返回一个纯文本响应（不被显示的）。
    抛出一个异常。

6. HttpResponse对象离开Django后，被渲染为一个字符串。

7. 在浏览器见到一个美化的，渲染后的web页面。
```

虽然某些细节被省略掉，这个解释应该有助于欣赏Django的高级架构。它也展示了关键的组件所扮演的角色，比如模型，视图，和模板。Django的很多组件都基于这几个广为人知设计模式。

什么是模式？

What is coon between the words Blueprint, caffolding, and Maintenance?
These software development terms have been borrowed from the world of building construction and architecture. However, one of the ost inuential ters coes from a treatise on architecture and urban planning written in 1977 by the leading Austrian architect Christopher Alexander and his team consisting of Murray Silverstein, Sara Ishikawa, and several others.

The term "Pattern" came in vogue after their seminal work, A Pattern Language: Towns, Buildings, Construction volue in a fivebook series based on the astonishing insight that users know about their buildings more than any architect ever could. A pattern refers to an everyday problem and its proposed but time-tested solution.

In the book, Christopher Alexander states that "Each pattern describes a problem, which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice."

For example, the Wings Of Light pattern describes how people prefer buildings with more natural lighting and suggests arranging the building so that it is composed of wings. These wings should be long and narrow, never more than 25 feet wide. Next time you enjoy a stroll through the long well-lit corridors of an old university, be grateful to this pattern.

Their book contained 253 such practical patterns, from the design of a room to the design of entire cities. Most importantly, each of these patterns gave a name to an abstract problem and together formed a pattern language.

Reeber when you first cae across the word déjà vu? You probably thought "Wow, I never knew that there was a word for that experience." Similarly, architects were not only able to identify patterns in their environent but could also, finally, name them in a way that their peers could understand.

In the world of software, the term design pattern refers to a general repeatable solution to a commonly occurring problem in software design. It is a formalization of best practices that a developer can use. Like in the world of architecture, the pattern language has proven to be extremely helpful to communicate a certain way of solving a design problem to other programmers.

There are several collections of design patterns but some have been considerably ore inuential than the others.

四人组模式

One of the earliest efforts to study and document design patterns was a book titled Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, who later became known as the Gang of Four (GoF). This book is so inuential that any consider the design patterns in the book as fundamental to software engineering itself.

In reality, the patterns were written primarily for object-oriented programming languages, and it had code examples in C++ and Smalltalk. As we will see shortly, many of these patterns might not be even required in other programming languages with better higher-order abstractions such as Python.

The patterns have been broadly classified by their type as follows :

- Creational Patterns: These include Abstract Factory, Builder Pattern, Factory Method, Prototype Pattern, and Singleton Pattern
- Structural Patterns: These include Adapter Pattern, Bridge Pattern, Composite Pattern, Decorator Pattern, Facade Pattern, Flyweight Pattern, and Proxy Pattern
- Behavioral Patterns: These include Chain of Responsibility, Command Pattern, Interpreter Pattern, Iterator Pattern, Mediator Pattern, Memento Pattern, Observer Pattern, State Pattern, Strategy Pattern, Template Pattern, and Visitor Pattern

While a detailed explanation of each pattern would be beyond the scope of this book, it would be interesting to identify some of these patterns in Django itself:

Django中的模式与此对比：

四人组模式	**Django**组件	解释
命令模式	HttpRequest	把一个request封装进一个对象
观察者模式	Signals	一个对象改变状态时，它的所有侦听器都被通知并自动更新
模板模式	基于类的视图	一个算法的步骤可以不用改变算法结构来重新定义子类

而这些模式是对于学习Django内部的人来说构造最有趣的，Django下面的模式可以再次分类——这也是个常见的问题。

Django是MVC吗？

Model-View-Controller (MVC) is an architectural pattern invented by Xerox PARC in the 70s. Being the framework used to build user interfaces in Smalltalk, it gets an early mention in the GoF book.

Today, MVC is a very popular pattern in web application frameworks. Beginners often ask the questionis Django an M fraework?

The answer is both yes and no. The MVC pattern advocates the decoupling of the presentation layer from the application logic. For instance, while designing an online game website API, you might present a game's high scores table as an HTML, ML, or coaseparated file. However, its underlying odel class would be designed independent of how the data would be finally presented.

MVC is very rigid about what models, views, and controllers do. However, Django takes a much more practical view to web applications. Due to the nature of the HTTP protocol, each request for a web page is independent of any other request. Django's framework is designed like a pipeline to process each request and prepare a response.

Django calls this the Model-Template-View (MTV) architecture. There is separation of concerns between the database interfacing classes (Model), request-processing classes iew, and a teplating language for the final presentation Teplate.

If you compare this with the classic MVC—"Model" is comparable to Django's Models, "View" is usually Django's Templates, and "Controller" is the framework itself that processes an incoming HTTP request and routes it to the correct view function.

If this has not confused you enough, Django prefers to name the callback function to handle each URL a "view" function. This is, unfortunately, not related to the MVC pattern's idea of a View.

MVC对于模型，视图，和控制该做什么设定的非常严格。然而，到web应用Django采取的是更实用的视图。要处理原生的HTTP协议，每个到web页面的请求独立于其他的请求。Django的框架设计的像一个管道处理每个请求，准备好响应。

Django称之为模型-模板-视图（MTV）架构。数据库接口类（模型）和 请求处理类（视图），以及最终表现的模板语言之间有着独立关系。

如果你将它于经典的MVC比较，“模型”可以通Django的模型比较，“视图”

要是这些都还不足以让你感到迷惑，Django倾向于选择命名回调函数处理每个URL的“视图”函数。即，一个没有MVC视图概念的视图。

福勒模式

In 2002, Martin Fowler wrote Patterns of Enterprise Application Architecture, which described 40 or so patterns he often encountered while building enterprise applications.

Unlike the GoF book, which described design patterns, Fowler's book was about architectural patterns. Hence, they describe patterns at a much higher level of abstraction and are largely programming language agnostic. Fowler's patterns are organized as follows:

• Domain Logic Patterns: These include Domain Model, Transaction Script, Service Layer , and Table Module • Data Source Architectural Patterns: These include Row Data Gateway, Table Data Gateway, Data Mapper, and Active Record • Object-Relational Behavioral Patterns: These include Identity Map, Unit of Work, and Lazy Load • Object-Relational Structural Patterns: These include Foreign Key Mapping, Mapping, Dependent Mapping, Association Table Mapping, Identity Field, Serialized LOB, Embedded Value, Inheritance Mappers, Single Table Inheritance, Concrete Table Inheritance, and Class Table Inheritance • Object-Relational Metadata Mapping Patterns: These include Query Object, Metadata Mapping, and Repository • Web Presentation Patterns: These include Page Controller, Front Controller, Model View Controller, Transform View, Template View, Application Controller, and Two-Step View • Distribution Patterns: These include Data Transfer Object and Remote Facade • Offline Concurrency Patterns: These include Coarse Grained Lock, Implicit Lock, Optimistic fine Lock, and Pessiistic fine Lock • Session State Patterns: These include Database Session State, Client Session State, and Server Session State • Base Patterns: These include Mapper, Gateway, Layer Supertype, Registry, Value Object, Separated Interface, Money, Plugin, Special Case, Service Stub, and Record Set

Almost all of these patterns would be useful to know while architecting a Django application. In fact, Fowler's website at http://martinfowler.com/eaaCatalog/ has an excellent catalog of these patterns. I highly recommend that you check them out.

Django also implements a number of these patterns. The following table lists a few of them:

Django中的模式与此对比：

四人组模式	**Django**组件	解释
活动记录	Django模型	封装数据库访问，对数据添加域名逻辑
类表继承	模型继承	继承中的每个实体都映射到一个独立的表
标识自动	Id字段	在一个对象中保存一个数据库ID字段以维护标识
模板视图	Django模板	使用HTML的内嵌生成器渲染到HTML

还有更多的模式？

Yes, of course. Patterns are discovered all the tie. Like living beings, soe mutate and form new patterns: take, for instance, MVC variants such as Model–view–presenter (MVP), Hierarchical model–view–controller (HMVC), or Model View ViewModel (MVVM).

Patterns also evolve with tie as better solutions to known probles are identified. For example, Singleton pattern was once considered to be a design pattern but now is considered to be an Anti-pattern due to the shared state it introduces, similar to using global variables. An Anti-pattern can be defined as coonly reinvented but a bad solution to a problem.

Some of the other well-known books which catalog patterns are Pattern-Oriented Software Architecture (known as POSA) by Buschmann, Meunier, Rohnert, Sommerlad, and Sta; Enterprise Integration Patterns by Hohpe and Woolf; and The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience by Duyne, Landay, and Hong.

本书中模式

This book will cover Djangospecific design and architecture patterns, which would be useful to a Django developer. The upcoming sections will describe how each pattern will be presented.

Pattern name The heading is the pattern name. If it is a well-known pattern, the commonly used name is used; otherwise, a terse, self-descriptive name has been chosen. Names are important, as they help in building the pattern vocabulary. All patterns will have the following parts:
Problem: This briey entions the proble.

Solution: This summarizes the proposed solution(s).

Problem Details: This elaborates the context of the problem and possibly gives an example.
Solution Details: This explains the solution(s) in general terms and provides a sample Django implementation.

本书覆盖针对对于Django开发者会很有用的Django的设计和架构模式。接下来的章节会描述每个模式是如何实现的。

模式名称 标题是模式名称。如果它是知名的模式，常用的名字被使用；否则，简洁的，自描述的名称被选择。名称非常重要，它们有助于构建模式词汇。所有的模式都有以下部分：

鉴审模式

Despite their near universal usage, Patterns have their share of criticism too. The most common arguments against them are as follows:

尽管它们近乎于通用，模式共享也有它们的审鉴共享。它们的最常见参数如下：

- Patterns compensate for the missing language features: Peter Norvig found that 16 of

the 23 patterns in Design Patterns were 'invisible or simpler' in Lisp. Considering Python's introspective facilities and firstclass functions, this ight as well be the case for Python too.
- Patterns repeat best practices: Many patterns are essentially formalizations of best practices such as separation of concerns and could seem redundant.
- Patterns can lead to over-engineering: Implementing the pattern might be less efficient and excessive copared to a sipler solution.

如何使用模式

While some of the previous criticisms are quite valid, they are based on how patterns are misused. Here is some advice that can help you understand how best to use design patterns:

- Don't implement a pattern if your language supports a direct solution
- Don't try to retrofit everything in ters of patterns
-
- Use a pattern only if it is the most elegant solution in your context
- 在开发环境中且仅当解决方法为最简练时使用模式
- Don't be afraid to create new patterns
- 不要害怕创建新模式

> 最佳实践
>
> In addition to design patterns, there might be a recommended approach to solving a problem. In Django, as with Python, there might be several ways to solve a problem but one idiomatic approach among those.
> 除了设计模式之外，还是存在其他推荐的方法来解决一个问题。在Django中，因为使用的语言就是Python，所以我们就有

Python之禅和Django的设计哲学

Generally, the Python community uses the term 'Pythonic' to describe a piece of idiomatic code. It typically refers to the principles laid out in 'The Zen of Python'. Written like a poem, it is extremely useful to describe such a vague concept.

通常来说，Python社区使用术语“Python范儿”来描述一段编写手法地道的代码。我们通常参照“Python之禅”中出现的原则。就像写诗歌一样，它在描述一个模糊的概念时及其有用。

> Try entering import this in a Python prompt to view 'The Zen of Python'.

Furthermore, Django developers have crisply documented their design philosophies while designing the framework at https://docs.djangoproject.com/en/dev/ misc/design-philosophies/.

While the document describes the thought process behind how Django was designed, it is also useful for developers using Django to build applications. Certain principles such as Don't Repeat Yourself (DRY), loose coupling, and tight cohesion can help you write more maintainable and idiomatic Django applications.

Django or Python best practices suggested by this book would be formatted in the following manner:

> 最佳实践：
>
> 在settings.py中使用BASE_DIR，避免硬编码目录名称。

总结

In this chapter, we looked at why people choose Django over other web frameworks, its interesting history, and how it works. We also examined design patterns, popular pattern collections, and best practices.

本章，我们浏览了为什么人们选择Django而不是其他的Web框架，以及它的发展史，和Django的工作原理。我们也验证了设计模式，流行的模式集合，以及最佳实践。

In the next chapter, we will take a look at the first few steps in the beginning of a Django project such as gathering requirements, creating mockups, and setting up the project.

在下一章，我们会学习Django项目中开始的头几个步骤，比如获取请求，模型建模，以及项目配置。

第二章 应用模式

本章，我们将学习以下内容：

- 获取请求
- 创建一个概念文档
- 如何将一个项目分为多个app
- 是重新写一个新的app还是使用已有的
- 开始一个项目之前的最佳实践
- 为什是Python3?
- 启动SuperBook项目

很多的开发新手从依照正确方法来写代码开始，完成一个新的项目。而更多的人经常被引到错误的假想，未被使用的功能，并且浪费掉很多时间。花些时间在你的客户身上，来理解一个项目中的核心请求事件，即使很短的时间也能产生惊人的效果。管理请求是一个值得学习的关键知识点。

如何获取请求

`创新不关乎肯定一切，而是关乎对所有重要的特性说不 ---Steve Jobs`

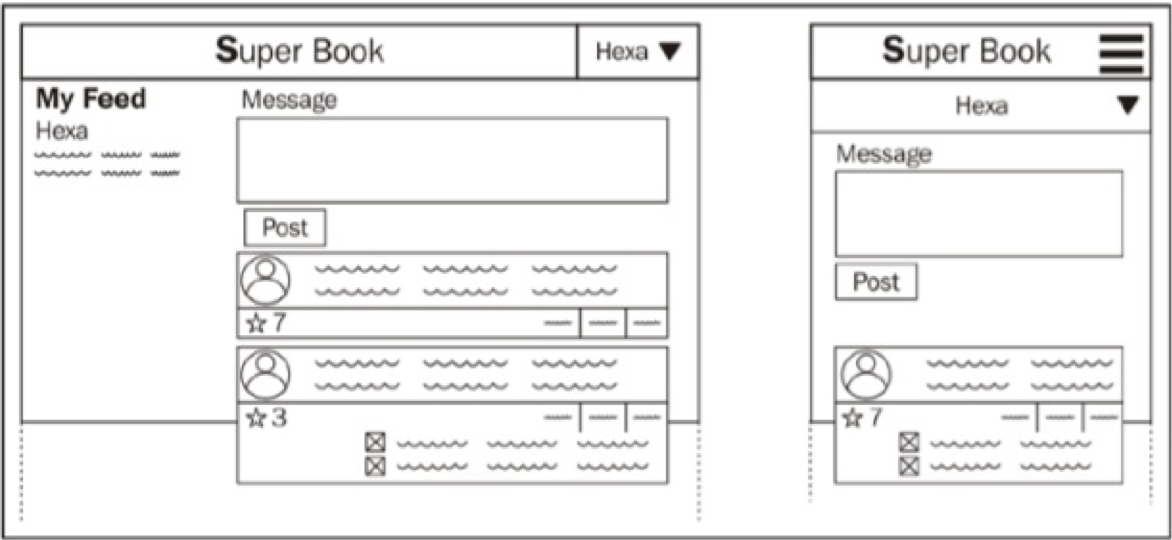
我通过与客户耗去数天来仔细地的倾听他们的需求，以及合理的期望值，拯救了好几个注定失败的项目。除了纸（或者他们的数字设备）和一支笔之外什么也没用，处理过程极其简单但是却很有效。这里是一些获取请求的关键地方：

```
1. 直接和应用的所有者沟通即使他们没有技术背景。
2. 确保你完整的倾听他们的需求并提醒他们。
3. 不要使用“模型”这一类的技术术语。保证用语简单，使用终端用户能理解的术语比如“用户账户”。
4.合理的期望值。如果某件事情在技术行不可行，或者很难实现，要保证你以正确的方法告知他们。
5.描述的尽可能具象。在大自然中人类是依靠视觉。网站更是如此。使用粗线条和图形表。不需要多么的完美。
6.分解处理流程，比如用户注册。任何多步骤的功能都需要以用箭头连接的方框画出。
7.最后，解决用户叙述表格中的特性，
8.扮演一个活动的角色
9.在融合新特性上要非常，非常地保守。
10.开会，和大家分享你的笔记以避免误解。
```

第一个会议会比较长（可能是一整个工作日，或者好几个小时）。之后，当这些会议的沟通变的顺畅时，你可以将它们削减到30分钟或者一个小时。

所有的这一切的结果会是写了满满的一整页，以及好几张的乱糟糟的草图。

在本书，我们已经着手建构我们自己的一个为超级英雄而构建的称做SuperBook的社会网络。A simple sketch based off our discussions with a bunch of randomly selected superheroes is shown as follows:



按照响应式设计的*SuperBook*网站草图。桌面（左边）和手机（右边）的效果如上.

你是一个会讲故事的人吗？

那么这一章写的是什么呢？这是一个解释使用这个网站是做何感觉的简易文档。

> ## 注释
>
> **SuperBook**的概念 下面的访问记录是在未来我们的网站SuperBook运营之后所做的整理。
>
> 注释
>
> **SuperBook**的理念 下面的是采访是在未来我们的网站SuperBook网站运行起来之后的记录。一个半小时的用户测试在采访之前就做过了。
>
> 请你作下自我介绍 俺叫阿克苏。俺是一个生活在纽市中心的灰松鼠。不过呢，大家都叫俺阿康。俺爹叫缇巴厘是一个嘻哈音乐明星，他过去也这样叫俺。俺猜啊，俺从来在唱歌方面就差好多，因此没能够子承父业。
>
> 实际哩，在俺很小的时候，俺就有点儿喜欢偷东西。医生说俺对坚果过敏，这也是大家伙都知道嘀。俺家其他的几个哥哥吃坚果都没啥事儿。俺几个兄弟都可以在任何地方活下去。俺也曾酒馆，电影院，游乐园，还有好多地方把歌唱。俺看唱片的时候，忒别认眞。
>
> 好吧，阿康。你为什么被挑选为测试用户，你知道为什么吗？
>
> 可能吧，因为俺是纽约大明星里最少知道超级英雄的人。俺猜啊，这人类啊都笑话俺这只可以用苹果电脑的松鼠。补充一句，俺可是一只非常有耐心的松鼠。
>
> 就你所见到的，你来说一说你所认识的**SuperBook**是个什么样子

> Actually, in my early days, I was a bit of a kleptomaniac. I am allergic to nuts, you know. Other bros have it easy. They can just live off any park. I had to improvise—cafes, movie halls, amusement parks, and so on. I read labels very carefully too.
>
> *Ok, Acorn. Why do you think you were chosen for the user testing?*
>
> Probably, because I was featured in a Y tar special on lesserknown superheroes. I guess people find it ausing that a squirrel can use a MacBook (Interviewer: this interview was conducted over chat). Plus, I have the attention span of a squirrel.
>
> *Based on what you saw, what is your opinion about SuperBook?*
>
> I think it is a fantastic idea. I mean, people see superheroes all the time. However, nobody cares about them. Most are lonely and antisocial. SuperBook could change that.
>
> *What do you think is different about Superbook?*
>
> It is built from the ground up for people like us. I mean, there is no "Work and Education" nonsense when you want to use your secret identity. Though I don't have one, I can understand why one would.
>
> *Could you tell us briefly some of the features you noticed?*
>
> Sure, I think this is a pretty decent social network, where you can:
>
> ```
> • Sign up with any user name (no more, "enter your real name", silliness)
> • Fans can follow people without having to add them as "friends"
> • Make posts, comment on them, and re-share them
> • Send a private post to another user
> ```
>
> Everything is easy. It doesn't take a superhuan effort to figure it out. Thanks for your time, Acorn.

为HTML版面设计

在构建web应用的前几天，像Photoshop和Flash这样的工具做到像素级别的模型效果使用是非常广泛的。 They are hardly recommended or used anymore.

现在对智能手机、平板、笔记本电脑和其他的平台保持原生和体验一致比像素层面的效果更为重要了。实际上，大多数的web设计者都基于HTML构建布局。

Creating an HTML mockup is a lot faster and easier than before. If your web designer is unavailable, developers can use a CSS framework such as Bootstrap or ZURB Foundation framework to create pretty decent mockups.

构建一个HTML版本相比以前快了很多。如果你暂时还没找到Web设计，那么开发者可以利用Bootstrap或者ZRUB这样的基本的CSS框架来构建非常漂亮的版面。

The goal of creating a mockup is to create a realistic preview of the website. It should not erely focus on details and polish to look closer to the final product compared to a sketch, but add interactivity as well. Make your static HTML come to life with working links and some simple JavaScript-driven interactivity.

构建版面的目的就是为了创建一个实际上可用网站的预览。

A good mockup can give 80 percent of customer experience with less than 20 percent of the overall development effort.

一个好的版面可以达到百分之八十的客户体验，以及少于百分之二十的总的开发成本。

设计应用

当你对于自己所要构建的东西有相当好的构思时，你可以开始思考如何使用Django来实现它。再者，这可以诱导人开始编程。然而，当你花了几分钟时间来思考设计时，你可以发现非常多的不同方法来解决一个设计问题。

一如测试驱动的开发方法论中提倡的那样，你也可以首先从测试开始。我们会在测试章节见到更多的TDD方法。我们会在测试那章见到更多的TDD方法。

Whichever approach you take, it is best to stop and think—"Which are the different ways in which I can ipleent this? What are the tradeoffs? Which factors are ore iportant in our context? Finally, which approach is the best?

不乱你选择的方式是哪一种，最好是停下来想一想——“”

Experienced Django developers look at the overall project in different ways. ticking to the DRY principle or soeties because they get lay, they think Have I seen this functionality before? For instance, can this social login feature be implemented using a third-party package such as django-all-auth? If they have to write the app themselves, they start thinking of various design patterns in the hope of an elegant design. However, they first need to break down a project at the top level into apps。

将一个项目分离为多个应用

Django的应用称做**project**。一个项目由多个应用或者**apps**组成。应用是一组拥有特定功能的Python包。

理论上说，每个app都必须是可以重复使用的。你可以按照自己的需要创建尽可能多的app。绝对不要害怕添加跟多的app，或者重构一个已经存在的应用到多个app。一个典型的Djanog项目包含15到20个app。

到了这一步，一个重要的决定就是是否使用Django的第三方应用，还是从零编写一个。第三方应用是无需你自己开发，便可以开箱即用的。大多数的包都可以很快安装并完成配置，几分钟之后就能够使用它们。

换句话来说，编写自己的应用意味着要常常自己设计并实现模型，视图，测试。Django和其他类型的应用并没有区别。

用自己写的，还是使用别人的

One of Django's biggest strengths is the huge ecosystem of third-party apps. At the time of writing, djangopackages.com lists ore than , packages. You ight find that your copany or personal library has even ore. nce your project is broken into apps and you know which kind of apps you need, you will need to take a call for each app—whether to write or reuse an existing one.

It might sound easier to install and use a readily available app. However, it not as simple as it sounds. Let's take a look at some third-party authentication apps for our project, and list the reasons why we didn't use them for SuperBook at the time of writing:

```
• Over-engineered for our needs: We felt that python-social-auth with support for any soc
• Too specific: Using django-facebook would mean tying our authentication to that provide
• Python dependencies: One of the requirements of django-allauth is python-openid, which
• Non-Python dependencies: Some packages might have non-Python dependencies, such as Redi
• Not reusable: Many of our own apps were not used because they were not very easy to reu
```

None of these packages are bad. They just don't meet our needs for now. They might be useful for a different project. In our case, the built-in Django auth app was good enough. On the other hand, you might prefer to use a third-party app for some of the following reasons:

```
• Too hard to get right Do your odel's instances need to for a tree? Use django-mptt for
• Best or recommended app for the job: This changes over time but packages such as django
• Missing batteries: Many feel that packages such as django-model-utils and django-extens
• Minimal dependencies: This is always good in my book
```

So, should you reuse apps and save tie or write a new custo app? I would recommend that you try a third-party app in a sandbox. If you are an intermediate Django developer, then the next section will tell you how to try packages in a sandbox.

我的app沙箱

From time to time, you will come across several blog posts listing the "must-have Django packages". However, the best way to decide whether a package is appropriate for your project is Prototyping.

Even if you have created a Python virtual environment for development, trying all these packages and later discarding them can litter your environment. So, I usually end up creating a separate virtual environment named "sandbox" purely for trying such apps. Then, I build a small project to understand how easy it is to use.

Later, if I am happy with my test drive of the app, I create a branch in my project using a version control tool such as Git to integrate the app. Then, I continue with coding and running tests in the branch until the necessary features are added. Finally, this branch will be reviewed and merged back to the mainline (sometimes called master) branch.

它是由哪个包组成的？

为了阐明过程，我们的SuperBook项目可以粗略地分解为下列app（没有全部列出）：

Authentication（内建django.auth): 这个app处理用户注册，登录，和登出。 **Accounts**（定制）：这个app提偶那个额外的用户账户信息。 **Posts**（定制）：这个app提供发表和回复功能 **Pows**（定制）：这个app跟踪有多少“碰”（支持或者喜欢） **Boostrap forms**（crispy-forms）：这个app处理表单布局和风格

这里，一个app已经被标记为从零（标记为“定制”）构建，或者我们要使用的第三方Django应用。随着，项目的发展，这些选项或许改变。不过，这对一个新手足够了。

在开始项目之前

While preparing a development environment, make sure that you have the following in place:

```
• A fresh Python virtual environment: Python 3 includes the venv module or you can instal
• Version control: Always use a version control tool such as Git or Mercurial. They are l
• Choose a project template: Django's default project template is not the only option. Ba
• Deployment pipeline: I usually worry about this a bit later, but having an easy deploym
```

SuperBook-少年来吧，接受你的任务

This book believes in a practical and pragmatic approach of demonstrating Django design patterns and the best practices through examples. For consistency, all our examples will be about building a social network project called SuperBook.

SuperBook focusses exclusively on the niche and often neglected market segment of people with exceptional super powers. You are one of the developers in a tea comprised of other developers, web designers, a marketing manager, and a project manager.

The project will be built in the latest version of Python (Version 3.4) and Django (Version 1.7) at the time of writing. Since the choice of Python 3 can be a contentious topic, it deserves a fuller explanation.

为什么选**Python 3**？

While the development of Python started in , its first release, Python ., was released on December 3, 2008. The main reasons for a backward incompatible version were—switching to Unicode for all strings, increased use of iterators, cleanup of deprecated features such as old-style classes, and some new syntactic additions such as the nonlocal statement.

The reaction to Python 3 in the Django community was rather mixed. Even though the language changes between Version 2 and 3 were small (and over time, reduced), porting the entire Django codebase was a significant igration effort.

n February , Django . becae the first version to support Python . Developers have clarified that, in future, Django will be written for Python with an ai to be backward compatible to Python 2.

For this book, Python 3 was ideal for the following reasons:

就本书来说，Python 3 由于以下几个原因看来它是最理想的：

• Better syntax This fixes a lot of ugly syntaxes, such as izip, xrange, and **unicode**, with the cleaner and more straightforward zip, range, and `__str__`.

• Sufficient third-party support: Of the top 200 third-party libraries, more than 80 percent have Python 3 support.

• No legacy code: We are creating a new project, rather than dealing with legacy code that needs to support an older version.

不存在遗留的代码：我们创建一个新的项目，而不是来处理需要支持旧版本代码。

• Default in modern platforms: This is already the default Python interpreter in Arch Linux. Ubuntu and Fedora plan to complete the switch in a future release.

• It is easy: From a Django development point of view, there are very few changes, and they can all be learnt in a few minutes.

使用方便：站在Django开发的观点来看，其改变非常微小，所有变更的地方学起来也很快。

The last point is important. Even if you are using Python 2, this book will serve you fine. Read appendix to understand the changes. You will need to make only minimal adjustments to backport the example code.

最后一点很重要。即使你现在使用的是Python 2，本书中的例子也能够很好的运行。

开始项目

本节是SuperBook项目的安装说明，它包含在本书中使用的所有代码实例。要为最新的安装注释检查项目的README文件。推荐你创建一个新的目录，superbook，首先包含虚拟环境，项目源码。

Ideally, every Django project should be in its own separate virtual environment. This makes it easy to install, update, and delete packages without affecting other applications. In Python 3.4, using the built-in venv module is recommended since it also installs pip by default:

理论上，每一个Django项目都应该有自己的虚拟环境。使用虚拟环境可以让Django的安装，升级显得很轻松，而且删除包也不会影响到其他的应用。在Python 3.4中，建议使用内建的venv模块，因为它默认也安装了pip：

```
$ python3 -m venv sbenv
$ source sbenv/bin/activate
$ export PATH="`pwd`/sbenv/local/bin:$PATH"
```

These commands should work in most Unix-based operating systems. For installation instructions on other operating systems or detailed steps please refer to the README file at the ithub repository   https://github.com/DjangoPatternsBook/ superbook. In the first line, we are invoking the Python . executable as python3; do confir if this is correct for your operating syste and distribution.

这些命令可以运行在大多数的基于Unix的操作系统。例如

The last export command might not be required in some cases. If running pip freeze lists your system packages rather than being empty, then you will need to enter this line.

代码中位于最后`export`命令在某些情况下不是必须的。如果运行`pip freeze`列出的系统包是空的，那么你需要输入这一行。

> 提示
>
> 开始Django项目之前，创建一个新的虚拟环境

接下来，从GitHub克隆项目例子，并安装依赖：

```
$ git clone https://github.com/DjangoPatternsBook/superbook.git
$ cd superbook
$ pip install -r requirements.txt
```

如果你想看一看已完成的SuperBook网站，只要运行migrate并启动测试服务器就行了：

```
$ cd final
$ python manage.py migrate
$ python manage.py createsuperuser
$ python manage.py runserver
```

In Django 1.7, the migrate command has superseded the syncdb command. We also need to explicitly invoke the createsuperuser command to create a super user so that we can access the admin.

在Django1.7中，migrate命令已经取代了syncdb命令。我们也需要明确地调用creteasuperuser命令来创建一个超级用户，这样我们就可以访问admin了。

You can navigate to http://127.0.0.1:8000 or the URL indicated in your terminal and feel free to play around with the site.

你可以浏览`http://127.0.0.1:8000`或者在终端指明URL，随便玩玩这个网站。

总结

Beginners often underestimate the importance of a good requirements-gathering process. At the same time, it is important not to get bogged down with the details, because programming is inherently an exploratory process. The most successful projects spend the right amount of time preparing and planning before development so that it yields the maximum benefits.

新手总是低估了一个优质的`请求—获取`流程的重要性。于此同时，重要的是不要被细节所束缚，因为从本质来说，编程就是一个探索的过程。很多成功的项目在开发之前都花掉了大量的时间来准备和计划，这样才能获得最佳效果。

We discussed many aspects of designing an application, such as creating interactive mockups or dividing it into reusable components called apps. We also discussed the steps to set up SuperBook, our example project.

我谈论了设计应用的很多方面，比如创建交互模型，或者是将它分割到多个称为app的可重复使用组件。同时我们也讨论了配置项目SuperBook的步骤。

In the next chapter, we will take a look at each component of Django in detail and learn the design patterns and best practices around them.

在下一章，我们会详细地浏览Django的每个组件，并学习与这些组件相关的最佳实践。

第三章 模型

本章，我们将讨论以下话题：

```
模型的重要性
类图表
模型结构模式
模型行为模式
迁移
```

M比V和C都更大

在Django中，模型是具有处理数据库的一种面向对象的方法的类。通常，每个类都引用一个数据库表，，每个属性都引用一个数据库列。你可以使用自动生成的API查询这些表。

模型是很多其他组件的基础。只要你有一个模型，你可以快速地推导模型admin，模型表单，以及所有类型的通用视图。在每个例子中，你都需要下一个行或是两行代码，这样可以让它看上去没有太多魔法。

模型也被用在更多超出你期望的地方。这书因为Django可以以多种方法运行。Django的一些切入点如下：

```
熟悉web请求-响应流程
Django的交互式shell
管理命令
测试脚本
异步任务队列比如Celery
```

在多数的这些例子中，模型模块要导入（作为django.setup()的一部分）。因此，最好保证模型远离任何不必要的依赖，或者导入任何的其他Django组件，比如视图。

简而言之，恰当地设计模型是很重要的事情。现在，让我们从SuperBook模型设计开始。

注释

自带午餐便当
*作者注释：SuperBook项目的进度会以这样的一个盒子出现。你可以跳过这个盒子，但是在web应用项目中的情况下，你缺少的是领悟，经验 。

史蒂夫和客户的第一周——超级英雄情报监控（简称为S.H.I.M）。简单来说，这是一个大杂烩。办公室是非常未来化的，但是不论做什么事情都需要上百个审核和签字。

作为Django开发者的领队，史蒂夫已经配置好了中型的运行超过两天的4台虚拟机。第二天的一个早晨，机器自己不翼而飞了。一个附近的清洁机器人说，机器被法务部们给带走了，他们要对未经审核的软件安装做出处理。

然而，CTO哈特给予史蒂夫了极大的帮助。他要求机器在一个小时之内完好无损地给还回去。他还对SuperBook项目做出了提前审核以避免将来可能出现的任何阻碍。

那个下午的稍晚些时候，史蒂夫给他带了一个午餐便当。身着一件米色外套和浅蓝色牛仔裤的哈特如约而至。尽管高出周围人许多，有着清爽面庞的他依旧那么帅气，那么平易近人。他问史蒂夫如果他之前是否尝试过构建一个60年代的超级英雄数据库。

”嗯，对的，是哨兵项目么？“，史蒂夫说道。”是我设计的。数据库看上去被设计成了一个条目-属性-值形式的模式，有些地方我考虑用反模式。可能，这些天他们有一些超级英雄属性的小想法。哈特几乎等不到听完最后一句，他压低嗓门道：“没错。是我的错。另外，他们只给了我两天来设计整个架构。他们这是在要我老命啊！”

听了这些，史蒂夫的嘴巴张的大大的，三明治也卡在口中。哈特微笑着道：“当然了，我还没有尽全力来做这件事。只要它成长为100万美元的单子，我们就可以多花点时间在这该死的数据库上了。SuperBook用它就能分分钟完事的，小史你说呢？”

史蒂夫微微点头称是。他从来没有想过在这么样的地方将会有上百万的超级英雄出现。

模型搜寻

这是在SuperBook中确定模型的第一部分。通常对于一个早期试验，我们只表示了基本模型，以及在一个类表中的表单的基本关系：

图：略

让我们忘掉模型一会儿，来谈谈我们正在构建的对象的术语。每个用户都有一个账户。用户可以写多个回复或者多篇文章。**Like**可以同时关联到一个单独的用户或者文章。

推荐你像这样画一个模型的类表。这个步骤的某些属性或许缺失了，不过你可以在之后补充细节。只要整个项目用图表表示出来，便可以轻松地分离app。

这是创建此表示的一些提示：

```
盒子表示条目，它将成为模型。
名词通常作为条目的终止。
箭头是双向的，表示Django中三种关系类型其中的一种：一对一，一对多（通过外键实现），和多对多。
字段表明在基于条目-关系模型（ER-modle）的模型中定义了一对多关系。换句来说，星号是声明外键的地方。
```

类图表可以映射到下面的Django代码中（它将遍及多个app）：

```
class Profile(models.Model):
    user = models.OnToOneField(User)

class Post(models.Model):
    posted_by = models.ForeignKey(User)

class Comment(models.Model):
    commented_by = models.ForeignKey(User)
    for_post = models.ForeignKey(Post)

class Like(models.Model):
    liked_by = models.ForeignKey(User)
    post = models.ForeignKey(Post)
```

这之后，我们不会直接地引用*User*，而是使用更加普通的*settings.AUTH_USER_MODEL*来替代。

分割model.py到多个文件

就像多数的Django组件那样，一个大的model.py文件可以在一个包内分割为多个文件。**package**通过一个目录来实现，它包含多个文件，目录中的一个文件必须是一个称为 `__init__.py` 特殊文件。

所有可以在包级别中暴露的定义都必须在 `__init__.py` 里使用全局变量域定义。例如，如果我们分割model.py到独立的类，models子文件夹中的对应文件，比如，postable.py，post.py和comment.py, 之后 `__init__.py` 包会像这样：

```
from postable import Postable
from post import Post
from commnet import Comment
```

现在你可以像之前那样导入models.Post了。

在 `__init__.py` 包中的任何其他代码都会在包运行时被导入。因此，它是一个任意级别包初始化代码的理想之地。

结构模式

本节包含多个帮助你设计和构建模型的设计模式。

模式-规范化模型

问题：通过设计，模型实例的重复数据引起数据不一致。

解决方法：通过规范化，分解模型到更小的模型。使用这些模型之间的逻辑关系来连接他们。

问题细节

想象一下，如果某人用下面的方法设计Post表（省略部分列）：

超级英雄的名字	消息	发布时间
Captain Temper	消息已经发布过了？	2012/07/07/07:15
Professor English	应该用“Is”而不是“Has"	2012/07/07/07:17
Captain Temper	消息已经发布过了？	2012/07/07/07:18
Capt. Temper	消息已经发布过了？	2012/07/07/07:19

我希望你注意到了在最后一行的超级英雄名字和之前不一致（船长一如既往的缺乏耐心）。

如果我们看看第一列，我们也不确定哪一个拼写是正确的——`Captain Temper或者Capt.Temper`。这就是我们要通过规范化消除的一种数据冗余。

详解

在我们看下完整的规范方案，让我们用Django模型的上下文来个关于数据库规范化的简要说明。

规范化的三个步骤

规范化有助于你更有效地的存储数据库。只要模型完全地的规范化处理，他们就不会有冗余的数据，每个模型应该只包含逻辑上关联到自身的数据。

这里给出一个简单的例子，如果我们规范化了Post表，我们就可以不模棱两可地引用发布消息的超级英雄，然后我们需要用一个独立的表来隔离用户细节。默认，Django已经创建了用户表。因此，你只需要在第一列中引用发布消息的用户的ID，一如下表所示：

用户**ID**	消息	发布时间
12	消息已经发布过了？	2012/07/07/07:15
8	应该用“Is”而不是“Has"	2012/07/07/07:17
12	消息已经发布过了？	2012/07/07/07:18
12	消息已经发布过了？	2012/07/07/07:19

现在，不仅仅相同用户发布三条消息的清楚在列，而且我们可以通过查询用户表找到用户的正确的名字。

通常来说，你会按照模型的完全规规范化表来设计模型，也会因为性能原因而有选择性地非规范化设计。在数据库中，**Normal Forms**是一组可以被应用于表，确保表被规范化的指南。一般我们建立第一，第二，第三规范表，尽管他们可以递增至第五规范表。

这接下来的例子中，我们规范化一个表，创建对应的Django模型。想象下有个名字称做“Sightings”的表格，它列出了某人第一次发现超级英雄使用能力或者特异功能。每个条目都提到了已知的原始身份，超能力，和第一次发现的地点，包括维度和经度。

名字	原始信息	能力	第一次使用记录（维度，经度，国家，时间）
Blitz	Alien	Freeze Flight	+40.75, -73.99; USA; 2014/07/03 23:12
Hexa	Scientist	Telekinesis Flight	+35.68, +139.73; Japan; 2010/02/17 20:15
Traveller	Billonaire	Time travel	+43.62, +1.45, France; 2010/11/10 08:20

上面的地理数据提取自http://www.golombek.com/locations.html.

第一规范表（1NF）

为了确认第一个规范表格，这张表必须含有：

```
多个没有属性（cell）的值
一个主键作为单独一列或者一组列（合成键）
```

让我们试着把表格转换为一个数据库表。明显地，我们的 `Power` 列破坏了第一个规则。

更新过的表满足第一规范表。主键（用一个 `*` 标记）是 `Name` 和 `Power` 的合并，对于每一排它都应该是唯一的。

Name*	Origin	Power*	Latitude	Longtitude	Country	Time

Blitz |Alien |Freeze|+40.75170 |-73.99420|USA|2014/07/03 23:12|
Blitz|Alien|Flight|+40.75170|-73.99420|USA|2013/03/12 11:30|
Hexa|Scientist|Telekinesis|+35.68330|+139.73330|Japan|2010/02/17 20:15|
Hexa|Scientist|Filght|+35.68330|+139.73330|Japan|2010/02/19 20:30|
Traveller|Billionaire|Time tavel|+43.61670|+1.45000|France|2010/11/10 08:20|

第二规范表

第二规范表必须满足所有第一规范表的条件。此外，它必须满足所有非主键列都必须依赖于整个主键的条件。

在之前的表，我们注意到 `Origin` 只依赖于超级英雄，即，`Name`。不论我们谈论的是哪一个 `Power`。因此，`Origin` 不是完全地依赖于合成组件- `Name` 和 `Power`。

这里，让我们只取出原始信息到一个独立的，称做 `Origins` 的表：

Name*	Origin

Blitz|Alien| Hexa|Scientist| Traveller|Billionaire|

现在 `Sightings` 表更新为兼容第二规范表，它大概是这个样子：

Name*	Power*	Latitude	Longtitude	Country	Time

Blitz |Freeze|+40.75170 |-73.99420|USA|2014/07/03 23:12|
Blitz||Flight|+40.75170|-73.99420|USA|2013/03/12 11:30|
Hexa|Telekinesis|+35.68330|+139.73330|Japan|2010/02/17 20:15|
Hexa|Filght|+35.68330|+139.73330|Japan|2010/02/19 20:30| Traveller|Time
tavel|+43.61670|+1.45000|France|2010/11/10 08:20|

第三规范表

在第三规范表中，比表格必须满足第二规范表，而且应该额外满足所有的非主键列都直接依赖整个主键，而且这些非主键列都是互相独立的这个条件。

考虑下 `Country` 类。给出 `维度` 和 `经度` ，你可以轻松地得出 `Country` 列。即使观测到超级英雄的地方依赖于 `Name-Power` 合成键，但是它只是间接地依赖他们。

因此，我们把详细地址分离到一个独立的国家表格中：

Location ID	Latitude*	Longtitude*	Country

1|+40.75170|-73.99420|USA| 2|+35.68330|+139.73330|Japan|
3|+43.61670|+1.45000|France|

现在 `Sightings` 表格的第三规范表大抵如此：

	User ID*	Power*	Location ID	Time
2	Freeze	1	2014/0703 23:12	
2	Flight	1	2013/03/12 11:30	
4	Telekinesis	2	2010/02/17 20:15	
4	Flight	2	2010/02/19 20:30	
7	Time tavel	3	2010/11/10 08:20	

如之前所做的那样，我们用对应的 `User ID` 替换了超级英雄的名字，这个用户ID用来引用用户表格。

Django模型

现在我们可以看看这些规范化的表格可以用来表现Django模型。Django中并不直接支持合成键。这里用到的解决方案是应用代理键，以及在 `Meta` 类中指定 `unique_together` 属性：

```
class Origin(models.Model):
    superhero = models.ForeignKey(settings.AUTH_USER_MODEL)
    origin = models.CharField(max_length=100)

class Location(models.Model):
    latitude = models.FloatField()
    longtitude = models.FloatField()
    country = models.CharField(max_length=100)

    class Meta:
        unique_together = ("latitude", "longtitude")

class Sighting(models.Model):
    superhero = models.ForeignKey(settings.AUTH_USER_MODEL)
    power = models.CharField(max_length=100)
    location = models.ForeignKey(Location)
    sighted_on = models.DateTimeField()

    class Meta:
        unique_together = ("superhero", "power")
```

性能和非规范化

规范化可能对性能有不利的影响。随着模型的增长，需要应答查询的连接数也随之增加。例如，要在美国发现具有冷冻能力的超级英雄的数量，你需要连接四个表格。先前的内容规范后，任何信息都可以通过查询一个单独的表格被找到。

你应该设计模式以保持数据规范化。这可以维持数据的完整性。然而，如果你面临扩展性问题，你可以有选择性地从这些模型取得数据以生成非规范化的数据。

提示

最佳实践 因设计而规范，又因优化而非规范 例如，在一个确定的国家中计算观测次数是非常普通的，然后将观测次数作为一个附加的字段到 `Location` 模型。现在，你可以使用Django ORM 继承其他的查询，而不是一个缓存的值。

然而，你需要在每次添加或者移除观测时更新这个计数。你需要添加本次计算到*Singhting*的 `save` 方法，添加一个信号处理器，甚至使用一个异步任务去计算。

如果你有一个跨越多个表的负责查询，比如国家的超能力计算，你需要创建一个独立的非规范表格。就像前面那样，我们需要在每一次规范化模型中的数据改变时更新这个非规范的表格。

令人惊讶的是非规范化在大型的网站中是非常普遍的，因为它是数度和存储空间两者之间的折衷。今天的存储空间已经比较便宜了，然而速度也是用户体验中至关重要的一环。因此，如果你的查询耗时过于久的话，那么就需要考虑非规范化了。

我们应该一直使用规范化吗？

过多的规范化是是件不必要的事。有时候，它可以引入一个非必需的能够重复更新和查询的表格。

例如，你的 `User` 模型或许有好多个家庭地址的字段，你可以规范这些字段到一个 `Address` 模型中。可是，多数情况下，把一个额外的表引进数据库是没有必要的。

与其针对大多数的非规范化设计，不如在代码重构之前仔细地衡量每个非规范化的机会，对性能和速度上做出一个折衷的选择。

模式-模型mixins

问题：明显地模型含有重复的相同字段/或者方法，违反了DRY原则。

方案：提取公共字段和方法到各种不同的可复用的模型mixins中。

问题细节

设计模型时，你或许某些公共属性或者行为跨模型类共享。例如， `Post` 和 `Comment` 模型需要一直跟踪自己的 `created` 日期和 `modified` 日期。手动地复制-粘贴字段和它们所关联的方法十分不符合DRY原则。

由于Django的模型是类，像合成以及继承这样的面向对象方法都是可以选择的解决方案。然而，合成（具有包含一个共享类实例的属性）需要一个额外的间接层访问字段。

继承是有技巧的。我们可以对 `Post` 和 `Comment` 使用一个公共基类。然而，在Django中有三种类型的继承：**concrete**（具体）, **abstract**（抽象）, 和**proxy**（代理）。

而具体继承的运行视派生基类而定，就像你在Python类中通常用到的那样。不过，在Django中，这个基类将被映射到一个独立的表中。每次你访问基本字段时，都需要一个明确的连接。这样做会带来非常糟糕的性能问题。

代理继承只能添加新的行为到父类。你不能够添加新字段。因此，这种情况下它的用处也不大。

最后，我们只有寄希望于抽象继承了。

详解

抽象基类是用于模型之间共享数据和行为的简洁方案。当你定义一个抽象类时，它在数据库中并没有创建任何与之对象的表。相反，这些字段是在派生出来的非抽象类中创建的。

访问抽象基类字段不需要 `JOIN` 语句。带有可管理字段的结果表格也是不解自明的。为了利用这些优点，大多数的Django项目都使用抽象基类实现公共字段或者方法。

抽象模型的局限在于：

- 它们不能够拥有外键或者其他模型的多对多字段。
- 它们不能够被实例化或者保存起来。
- 它们查询中不能够直接地使用，因为它没有管理器。

下面展示了post和comment类如何使用一个抽象基类进行初始设计：

```
class Postable(models.Model):
    created = models.DateTimeField(auto_now_add=True)
    modified = modified.DateTimeField(auto_now=True)
    message = models.TextField(max_length=500)

    class Meta:
        abstract = True


class Post(Postable):
    ...


class Comment(Postable):
        ...
```

要将一个模型转换到抽象基类，你需要在它的内部 `Meta` 类中写上 `abstract = True` 。这里的 `Postable` 是一个抽象基类。可是，它不是那么的可复用。

实际上，如果有一个类含有 `created` 和 `modified` 字段，我们在后面就可以在几乎任何需要时间戳的模型中重复使用这个时间戳功能。

模型mixins

模型mixins是一个可以把抽象类当作父类来添加的模型。不像其他的语法，比如Java那样，Python支持多种继承。因此，你可以列出一个模型的任意数量的父类。

Mixins应该是互相垂直的而且易于组合的。把一个mixin放进基类的列表，这些mixin应该可以正常运行。这样看来，它们在行为上更类似于合成而非继承。

小一些mixin的会好很多。不论何时当一个mixin变得很大，而且又违反了独立响应原则，就要考虑把它重构到一个小一些的类中去。就让mixin一次做好一件事吧。

在前面的例子中，用于更新 `created` 和 `modified` 的时间的模型mixin可以轻松地分解出来，一如下面代码所示：

```
class TimeStampedModel(models.Model):
    created = modified.TimeStampModel(auto_now_add=True)
    modified = modified.DateTimeField(auto_now=True)

    class Meta:
        abstract = True


class Postable(TimeStampedModel):
    message = models.TextField(max_length=500)
    ...

    class Meta:
        abstract = True


class Post(Postable):
    ...


class Comment(Postable):
    ...
```

我们现在有两个超类了。不过，功能之间显然都是独立的。mixin可以分离到自己的模块之内，或者在其他的上下文中被重复利用。

模式-用户账户

问题：每一个网站都存储一组不同的用户账户细节。然而，Django的内建 `User` 模型旨在针对认证细节。

方案：用一对一关系的用户模型，创建一个用户账户类。

问题细节

Django提供一个开箱即用的相当不错的**User**模型。你可以在创建超级用户或者登录amdin接口的时候用到它。它含有少量的基本字段，比如全名，用户名，和电子邮件。

然而，大多数的现实世界项目都保留了很多关于用户的信息，比如他们的地址，喜欢的电影，或者它们的超能力。打Django1.5开始，默认的User模型就可以被扩展或者替换掉。不过，官方文档极力推荐只存储认证数据，即便是在定制的用户模型中也是如此（毕竟，用户模型也是所属于 `auth` 这个app的）。

某些项目是需要多种类型的用户的。例如，SuperBook可以被超级英雄和非超级英雄所使用。这里或许会有一些公共字段，以及基于用户类型的不同字段。

详解

官方推荐解决方案是创建一个用户账户模型。它应该和用户模型有一个一对一的关系。其余的全部用户信息都存储于该模型：

```
class Profile(models.Model):
    user = models.OnToOneField(settings.AUTH_USER_MODEL, primary_key=True)
```

这里建议你明确的将 `primary_key` 赋值为 `True` ，以阻止类似PostgreSQL这样的数据库后端中的并发问题。剩下的模型可以包含其他的任何用户详情，比如生日，喜好色彩，等等。

设计账户模型之时，建议所有的账户详情字段都必须是非空的，或者含有一个默认值。凭直觉我们就知道用户在注册时是不可能填写完所有的账户细节的。此外，我们也要确保创建账户实例时，信号处理器没有传递任何初始参数。

信号

理论上，每一次用户模型实例的生成，其对应的用户账户实例也必须创建好。这个操作通常使用信号来完成。

例如，我们可以使用下面的信号处理器侦听用户模型的 `post_save` 信号：

```
# signals.py
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.conf import settings
from . import models


@receiver(post_save, sender=settings.AUTH_USER_MODEL)
def create_profile_handler(sender, instance, created, **kwargs):
    if not created:
        return
    # 仅在created是最新时才创建账户对象
    profile = models.Profile(user=instance)
    profile.save()
```

注意账户模型除了用户实例之外并没有传递额外初始的参数。

前面的代码中，初始化信号代码并没有放到特定的场所。通常，它们在 `models.py` 中（这样做是不可靠的）导入或者执行。不过，随着Django 1.7的应用载入的重构，应用初始化代码位置的问题也很好的解决了。

首先，为你的应用创建一个 `__init__.py` 包以引用应用的 `ProfileConfig` :

```
    default_app_config = "profile.apps.ProfileConfig"
```

接下来是 `app.py` 中的子类 `ProfileConfig` 方法，可使用 `ready` 方法配置信号：

```
# app.py
from django.apps import AppConfig


class ProfileConfig(AppConfig):
    name = "profiles"
    verbose_name = "User Profiles"


    def ready(self):
        from . import signals
```

随着信号的配置，对所有的用户来说，访问 `user.profile` 应该都返回一个 `Profile` 对象，即使是最新创建的用户也是如此。

Admin

现在，用户的详情存在admin内的两个不同地方：普通用户admin页面中的认证细节，同一个用户的额外账户细节放到了一个独立账户的admin页面， 这样做显得非常麻烦。

如下，为了操作方便，账户admin可以通过定义一个自定义的 `UserAdmin` 嵌入到默认的用户admin中：

```
# admin.py
from django.contrib import admin
from .models import Profile
from django.contrib.auth.models import User


class UserProfileInline(admin.StackedInline):
    model = Profile


class UserAdmin(admin.UserAdmin):
    inlines = [UserProfileInline]


admin.site.unregister(User)
admin.site.register(User, UserAdmin)
```

多个账户类型

假设在应用中你需要几种类型的用户账户。这里需要有一个字段去跟踪用户使用的是哪一种账户类型。账户数据本身需要存储在独立的模型中，或者存储在一个统一的模型中。

建议使用聚合账户的办法，因为它能够改变账户类型而不丢失账户细节，兼具灵活性，最小化复杂性。此办法中，账户模型包含一个所有账户类型的字段超集。

例如，SuperBook会需要一个 `SuperHero` 类型账户，和一个 `Ordinary` （非超集英雄）账户。它可以用一个独立的统一账户模型实现：

```
class BaseProfile(models.Model):
    USER_TYPES = (
        (0, 'Ordinary'),
        (1, 'SuperHero'),
    )

    user = models.OnToOneField(settings.AUTH_USER_MODEL, primary_key=True)
    user_type = models.IntegerField(max_length=1, null=True,    choices=USER_TYPES)
    bio = models.CharField(max_length=200, blank=True, null=True)

    def __str__(self):
        return "{}:{:.20}".format(self.user, self.bio or "")

    class Meta:
        abstract = True


class SuperHeroProfile(models.Model):
    origin = models.CharField(max_length=100, blank=True, null=True)

    class Meta:
        abstract = True


class OrdinaryProfile(models.Model):
    address = models.CharField(max_length=200, blank=True, null=True)

    class Meta:
        abstract = True


class Profile(SuperHeroProfile, OrdinaryProfile, BaseProfile):
    pass
```

我们把账户细节组织到多个抽象基类再到独立的关系中。`BaseProfile` 类包含所有的不关心用户类型的公共账户细节。它也有一个 `user_type` 字段，它持续追踪用户的激活账户。

`SuperHeroProfile` 类和 `OrdinaryProfile` 类分别包含所有到超级英雄和非超级英雄特定账户细节。最后，所有这些基类的 `profile` 类创建了一个账户细节的超集。

使用该方法时要主要的一些细节：

- 所有属于类的字段或者它抽象基类都必须是非空的，或者有一个默认值。
- 此方法或许会因为每个用户而消耗掉更多的数据库，但是却带来极大的灵活性。
- 账户类型的激活和非激活字段都必须在模型外部是可管理的。
- 说到，编辑账户的表必须显示合乎目前激活用户类型的字段。

模式-服务模式

问题：模型会变得庞大而且不可管控。当一个模型不止实现一个功能时，测试和维护也会变得困难。

解决方法：重构出一组相关方法到一个专用的 `Service` 对象中。

问题细节

富模型，瘦视图是一个通常要告诉Django新手的格言。理论上，你的视图不应该包含任何其他表现逻辑。

可是，随着时间的推移，代码段不能够放在任意地点，除非你打算将它们放进模型中。很快，模型会变成一个代码的垃圾场。

下面是模型可以使用 `Service` 对象的征兆：

```
1. 与扩展能服务交互，例如web服务中，检查一个用户具有资格。
2. 辅助任务不会处理数据库，例如，生成一个短链接，或者针对用户的验证码。
3. 牵涉到一个短命的对象时不会存在数据库状态记录，烈日，创建一个AJAX调用的JSON响应。
4. 对于长时间运行的任务设计到了多实例，比如Celery任务。
```

Django中的模型遵循着激活记录模式。理论上，它们同时封装应用罗即数据库访问。不过，要记得保持应用逻辑最小化。

在测试时，如果我们发现没有对数据库建模的必要，甚至不会用到数据库，那么我们需要考虑把分解模型类。建议这种场合下使用 `Service` 对象。

详解

服务对象是封装一个 `服务` 或者和系统狡猾的普通而老旧的Python对象（POPOs）。它们通常保存在一个独立的称为 `service.py` 或者 `utils.py` 的文件。

例如，像下面这样，检查一个web服务是否作为一个模型方法：

```
class Profile(models.Model):
    ...

    def is_superhero(self):
        url = "http://api.herocheck.com/?q={0}".format(
            self.user.username
            )
        return webclient.get(url)
```

该方法可以使用一个服务对象来重构：

```
from .services import SuperHeroWebAPI

def is_superhero(self):
    return SuperHeroWebAPI.is_superhero(self.user.username)
```

现在服务对象可以定义在 `services.py` 中了：

```
API_URL = "http://api.herocheck.com/?q={0}"

class SuperHeroWebAPI:
    ...

    @staticmehtod
    def is_hero(username):
        url = API_URL.format(username)
        return webclient.get(url)
```

多数情况下，`Service` 对象的方法是无状态的，即，它们基于函数参数不使用任何的类属性来独立执行动作。因此，最好明确地把它们标记为静态方法（就像我们对 `is_hero` 所做的那样）。

可以考虑把业务逻辑和域名逻辑从模型迁移到服务对象中去。这样，你可以在Django应用的外部很好使用它们。
想象一下，由于业务的原因，要依据某些用户的名字把这些要成为超级英雄的用户加进黑名单。我们的服务对象稍微改动以下就可以支持这个功能：

```
class SuperHeroWebAPI:
    ...

    @staticmethod
    def is_hero(username):
        blacklist = set(["syndrome", "kcka$$", "superfake"])
        url = API_URL.format(username)
        return username not in blacklist and webclient.get(url)
```

理论上，服务对象自包含的。这使它们易于测试而不用建模——即数据库，同时它们也轻松地重复使用。

Django中，耗时服务以Celery这样的异步任务队列方式执行。通常，`Service` 对象以Celery任务的方式执行操作。这样的任务可以周期性地运行或者延迟运行。

检索模式

本节包含处理模型特性的访问，或者对模型执行查询的设计模式。

模式-属性字段

问题：模型的属性以方法实现。可是，这些属性不应该保存到数据库。

**解决方案：对这样的方法使用特性装饰器。

问题详情

模型字段存储每个实例的属性，比如名，和姓，生日，等等。它们存储于数据库之中。可是，我们也需要访问某些派生的属性，比如一个完整的名字和年龄。

它们可以轻易地计算数据库字段，因此它们不需要单独地存储。在某些情况下，它们可以成为一个检查给出的年龄，会员积分，和激活状态是否合格的条件语句。

简洁明了的实现这个方法是定义类似下面的 `get_age` 来实现：

```
class BaseProfile(models.Model):
    birthdate = models.DateField()
    #...

    def get_age(self):
        today = datetime.date.today()
        return (today.year - self.birthdate.year) - int(
            (today.month, today.day) < (self.birthdate.month, self.birthdate.day)
            )
```

调用 `profile.get_age()` 便会通过计算调整过的月和日期所属的那个年份的不同来返回用户的年龄。

不过，调用 `profile.age` 更具可读性（和Python范）。

详解

Python类可以使用 `property` 装饰器把函数当作一个属性来使用。这样，Django模型也可以较好地利用它。替换前面那个例子中的函数：

```
    @property
    def age(self):
```

现在我们可以用 `profile.age` 来访问用户的年龄。注意，函数的名称要尽可能的短。

就像模型的方法那样，属性的一个重大缺陷是它对于ORM来说是不可访问的。你不能够在 `Queryset` 对象中使用它。例如，这么做是无效的，`Profile.objects.exlude(age__lt=18)。

它也是一个定义一个属性来隐藏类内部细节的好主意。这也正式地称做得墨忒耳定律。简单地说，定律声明你应该只访问自己的直属成员或者“仅使用一个点号”。

例如，最好是定义一个 `profile.birthyear` 属性，而不是访问 `profile.birthdate.year` 。这样，它有助于你隐藏 `birthdate` 字段的内在结构。

> # 提示
>
> 最佳实践
> 遵循得墨忒耳定律，并且访问属性时只使用点号

该定律的一个不良反应是它导致在模型中有多个包装器属性被创建。这使模型膨胀并让它们变得难以维护。利用定律来改进你的模型API，减少模型间的耦合，在任何地方都是可行的。

缓存特性

每次我们调用一个属性时，就要重新计算函数。如果计算的代价很大，我们就想到了缓存结果。因此，下次访问属性，我们就拿到了缓存的结果。

```
from django.utils.function import cached_property
    #...
    @cached_property
    def full_name(self):
        # 代价高昂的操作，比如，外部服务调用
        return "{0} {1}".format(self.firstname, self.lastname)
```

缓存的值会作为Python实例的一部分而保存。只要实例一直存在，就会得到同样的返回值。

就保护性机制来说，你或许想要强制执行代价高昂的操作以确保过期的值不会返回。这样，设置一个 `cached=False` 这样的关键字参数以阻止返回缓存的值。

模式-定制模型管理器

问题：某些模型的定义的查询被重复地访问，整个代码也就违反了DRY原则。

解决方案：通过定义自定义的管理器，使常见的查询拥有意义的名称。

问题细节

每一个Django的模型都有一个默认称做 `objects` 的管理器。调用 `objects.all()` 会返回数据库中的这个模型的所有条目。通常，我们只对所有条目的子集感兴趣。

我们应用多种过滤器以找出所需的条目组。挑选它们的原则常常是我们的核心业务逻辑。例如，我们发现使用下面的代码可以通过public访问文章：

```
    public = Posts.objects.filter(privacy="public")
```

这个标准在未来或许会改变。我们或许也想要检查文章是否标记为编辑。这个改变或许如此：

```
    public = Posts.objects.filter(privacy=POST_PRIVACY.Public, draft=Flase)
```

可是，这个改变需要使在任何地方都要用到公共文章。这令人非常沮丧。这仅需要一个定义这样常见地查询而无需“自我重复”。

详解

`Querysets` 是一个功能极其强大的抽象概念。它们仅在需要时才进行惰性查询。因此，通过链式方法（一个顺畅的接口）构建长的 `Querysets` 并不影响性能。

事实上，随着更多的过滤的应用反倒会使结果数据集合得以缩减。这样做的话通常能够减少内存消耗。

模型管理器是一个模型获取自身 `Queryset` 对象的便利接口。换句话来讲，它们有助于你使用Django的ORM访问下层的数据库。事实上，`QuerySet` 对象上管理器以一个非常简单的包装器实现。请注意相同到接口：

```
    >>> Post.objects.filter(posted_by__username="a")
    [<Post:a: Hello World>， <Post:a: This is Private!>]

    >>> Post.objects.get_queryset().filter(posted_by__username="a")
    [<Post:a: Hello World>, <Post:a: This is Private!>]
```

默认的管理器由Django创建，`objects` 有多种方法返回 `Queryset`，比如 `all`，`filter` 或者 `exclude`。不过，它们仅仅是生成了一个到数据库的低级API。

定制管理器用于创建特定的域名，高级API。这样不仅更具可读性，而且通过实现细节减轻所受到的影响。因此，你就能够利用高级抽象来严格的对域名建模了。

如下，前面的公开文章例子就可以很轻松地转换为一个定制的管理器：

```
# managers.py
from django.db.models.query import Queryset


class PostQuerySet(QuerySet):
    def public_posts(self):
        return self.filter(privacy="public")

PostManager = PostQuerySet.as_manager
```

这是一个在Django 1.7中从 `QuerySet` 对象创建定制管理器的捷径。不像前面的其他方法，这个 `PostManager` 对象像默认的 `objects` 管理器一样是可链式的。

如下所示，有些时候，使用定制的管理器替换去默认的 `objects` 管理器也是可行的：

```
from .managers import PostManager

class Post(Postable):
    ...
    objects = PostManager()
```

这样，访问 `public_posts` 就相当简单了：

```
    public = Post.objects.public_posts()
```

因为返回值是一个 `QuerySet`，它们可以更进一步过滤：

```
public_apology = Post.objects.public_posts().filter(
    message_startwith = "Sorry"
    )
```

`QuerySets` 由多个值得注意的属性。在下一节里，我们可以看到一些含有混合 `QuerySet` 的常见地模式。

Querysets的组合动作

事实上，对于它们的名字（或者是他们名字的后一半），`QuerySets` 支持多组（数学上的）操作。为了说明，考虑包含用户对象的两个 `QuerySets` ：

```
>>> q1 = User.objects.filter(username__in["a", "b", "c"])
[<User:a>, <User:b>, <User:c>]
>>> q2 = User.objects.filter(username__in["c", "d"])
[<User:c>, <User:d>]
```

对于一些组合操作可以执行以下动作：

- *Union*－交集：合并，移除重复动作。使用 `q1` | `q2` 获得 [`<User: a>, <User: b>, <User: c>, <User: d>`]
- *Intersection*－并集：找出公共项。使用 `q1` | `q2` 获得[`]
- *Difference*－补集：从第一个集合中移除同时包含在第二个集合中的元素。该操作并不按逻辑来。改用`q1.exlude(pk__in=q2)获得〔, 〕

同样的操作我们也可以用 `Q` 对象来完成：

```
from django.db.models import Q

# Union 交集
>>> User.objects.filter(Q(username__in["a", "b", "c"]) | Q(username__in=["c", "d"]))
[`<User: a>, <User: b>, <User: c>, <User: d>`]

# Intersection 并集
>>> User.objects.filter(Q(username__in["a", "b", "c"]) & Q(username__in=["c", "d"]))
[<User: c>]

# Difference 补集
>>> User.objects.filter(Q(username__in=["a", "b", "c"]) & ~Q(username__in=["c", "d"]))
[<User: a>, <User: b>]
```

注意执行动作所使用 `&` （和）以及 `~` （否定）的不同。`Q` 对象是非常强大的，它可以用来构建非常复杂的查询。

不过，`Set` 虽相似但却不完美。`QuerySets` 不像数学上的集合那样按照顺序来。因此，就这方面来说它们更接近于Python的列表数据结构。

链接多个Querysets

目前为止，我们已经合并了属于相同基类的同类型 `QuerySets` 。可是，我们或许需要合并来自不同模型的 `QuestSets` ，并对它们执行操作。

例如，一个用户的活动时间表包含了它们自身所有的按照反向时间顺序所发布的文章和评论。之前混合 `QuerySets` 方法是不会起作用的。一个很天真的做法是把它们转换到列表，连接并排列这个列表：

```
>>> recent = list(posts)+list(comments)
>>> sorted(recent, key=lambda e: e.modified, reverse=True)[:3]
[<Post: user: Post1>, <Comment: user: Comment1>, <Post: user: Post0>]
```

不幸的是，这个操作已经对惰性的 `QuerySets` 对象求值了。两个列表的内存使用算在一起可能很大内存开销。另外，转换一个庞大的 `QuerySets` 到列表是很慢很慢的。

一个更好的解决方案是使用迭代器减少内存消耗。如下，使用 `itertools.chain` 方法合并多个 `QuerySets` ：

```
>>> from itertools import chain
>>> recent = chain(posts, comments)
>>> sorted(recent, key=lambda e: e.modified, reverse=True)[:3]
```

只要计算 `QuerySets` ，连接数据的开销都会非常搞。因此，重要的是，尽可能长的仅有的不对 `QuerySets` 求值的操作时间。

> ## 提示
>
> 尽量延长 `QuerySets` 不求值的时间。

迁移

迁移让你改变模型时更有信心。说的Django 1.7，迁移已经是开发流程中基本的易于使用的一部分了。

新的基本流程如下：

```
1. 第一次定义模型类的话，你需要运行：

    python manage.py makemigrations <app_label>

2. 这将在`app/migrations/`文件夹内创建迁移脚本。
在同样的（开发）环境中运行以下命令：

    python manage.py migrate <app_label>

3. 这将对数据库应用模型变更。有时候，遇到的问题有，处理默认值，重命名，等等。

4. 普及迁移脚本到其他的环境。通常，你的版本控制工具，例如，Git，会小心处理这事。当最新的源释出时，新的迁移

5. 在这些环境中运行下面的命令以应用模型的改变：

    python manage.py migarte <app_label>

不论何时要将变更应用到模型，请重复以上1-5步骤。
```

如果你在命令里忽略了app标签，Django会在每一个app中发现未应用的变更并迁移它们。

总结

模型设计要正确地操作很困难。它依然是Django开发的基础。本章，我们学习了使用模型时多个常见模式。每个例子中，我们都见到了建议方案的作用，以及多种折衷方案。

这下一章，我们会用视图和URL配置来验证所遇到的常见设计模式。

第四章-视图和URL

本章，我们会讨论以下话题：

- 基于类的和基于函数的视图
- Mixins
- 装饰器
- 常见视图模式
- 设计URL

顶层的视图

Django中，视图是可以调用的，它接受请求并返回响应。通常它是一个函数或者是一个拥有 `as_view()` 这类特殊方法的类。

这两种情况下，我们创建一个普通的接受 `HTTPRequest` 作为自己的第一个参数并返回一个 `HTTPResponse` 的Python函数。 `URLConf` 也可以对这个函数传递额外的参数。这些参数由URL部分捕捉到，或者是设置了默认值。

这里是简单视图的例子：

```
# In views.py
from django.http import HttpResponse


def hello_fn(request, name="World"):
    return HttpResponse("Hellp {}!".format(name))
```

这两行视图函数非常简单和好理解。目前我们还没有用 `request` 参数来做任何事情。例如，通过查看 `GET/POST` 参数， URI路径，或者 `REMOTE_ADDR` 这样的HTTP头部，通过验证请求可以我们更好地理解所调用视图中的上下文。

`URLConf` 中所对应的行如下：

```
# In urls.py
    url(r'^hello-fn/(?P<name>\w+)/$', views.hello_fn),
    url(r'^hello_fn/$', views.hello_fn),
```

我们重复使用相同的视图以支持两个URL模式。第一个模式获得了一个name参数。第二个模式没有从URL获得任何参数，这个例子中视图会使用默认的 `World` 名字。

让视图变得更高级

基于类的视图在Django 1.4中被引入。下面是之前的视图在用了同等功能的基于类的视图重写之后的样子：

```
from django.views.generic import View


class HelloView(View):
    def get(self, request, name="World"):
        return HttpResponse("Hello {}!".format(name))
```

同样地，对应的 `URLConf` 也有两行，一如下面命令所示：

```
# In urls.py
    url(r'^hello-cl/(?P<name>\w+)/$', views.HelloView.as_view()),
    url(r'^hello-cl/$', views.HelloView.as_view()),
```

这个 `view` 类和我们之前的视图函数之间有多个有趣的不同点。最明显的一点就是我们需要定义一个类。接着，我们明确地定义了我们唯一要处理的 `GET` 请求。之前的视图对 `GET` ， `POST` 做出了同样的响应，或者其他的HTTP词汇，下面是在Django shell中使用测试客户端：

```
>>> from django.test import Client
>>> c = Client()
>>> c.get("http://0.0.0.0:8000/hello-fn/").content
b'Hello World!'
>>> c.post("http://0.0.0.0:8000/hello-fn/").content
b'Hello World!'
>>> c.get("http://0.0.0.0:8000/hello-cl/").content
b'Hello World!'
>>> c.post("http://0.0.0.0:8000/hello-cl/").content
b''
```

就安全和可维护性的角度来说，还是显式更好一些。

当你需要定制视图的时候，使用类的好处才会清晰的体现出来。就是说，你需要改变所问候的内容。你可以编写一个任意类型的普通视图类，并派生出所指定的问候类：

```
class CreetView(View):
    greeting = "Hello {}!"
    default_name = "World"
    def get(self, request, **kwargs):
        name = kwargs.pop("name", self.default_name)
        return HttpResponse(self.greeting.format(name))


class SuperVillianView(GreetView):
    greeting = "We are the future, {}. Not them."
    default_name = "my friend"
```

现在， `URLConf` 可以引用派生的类：

```
# In urls.py
    url(r'^hello-su/(?<name>\w+)/$', views.SuperVillianView.as_view()),
    url(r'^hello-su/$', views.SuperVillianView.as_view()),
```

按照类似的方式来定制视图函数不可行时，你需要添加多个有默认值的关键字参数。这样做很快会变得不可控制。这就是为什么通用视图从视图函数迁移到基于类的视图的原因。

> **Django Unchained**
>
> After spending 2 weeks hunting for good Django developers, Steve started to think out of the box. Noticing the tremendous success of their recent hackathon, he and Hart organized a Django Unchained contest at S.H.I.M. The rules were simple—build one web application a day. It could be a simple one but you cannot skip a day or break the chain. Whoever creates the longest chain, wins.
>
> The winner—Brad Zanni was a real surprise. Being a traditional designer with hardly any programming background, he had once attended week-long Django training just for kicks. He managed to create an unbroken chain of 21 Django sites, mostly from scratch.
>
> The very next day, Steve scheduled a 10 o' clock meeting with him at his office. Though Brad didn't know it, it was going to be his recruitment interview. At the scheduled time, there was a soft knock and a lean bearded guy in his late twenties stepped in.
>
> As they talked, Brad made no pretense of the fact that he was not a programmer. In fact, there was no pretense to him at all. Peering through his thick-rimmed glasses with calm blue eyes, he explained that his secret was quite simple—get inspired and then focus. He used to start each day with a simple wireframe. He would then create an empty Django project with a Twitter bootstrap template. He found Django's generic class-based views a great way to create views with hardly any code. Sometimes, he would use a mixin or two from Django-braces. He also loved the admin interface for adding data on the go.
>
> His favorite project was Labyrinth—a Honeypot disguised as a baseball forum. He even managed to trap a few surveillance bots hunting for vulnerable sites. When Steve explained about the SuperBook project, he was more than happy to accept the offer. The idea of creating an interstellar social network truly fascinated him.
>
> With a little more digging around, Steve was able to find half a dozen more interesting profiles like Brad within S.H.I.M. He learnt that rather that looking outside he should have searched within the organization in the first place.

基于类的通用视图

通常，基于类的通用视图为了能更好地重复使用代码，便利用以面向对象的方法（模板方法模式）实现的视图。我讨厌术语`generic views`。我更乐意把它们叫做`stock view`。就像所储备的照片，你可以按照自己的常见需要对它们做出轻微的调整。

通用视图的产生是因为Django开发者发现他们在每一个项目中都在重复构建同类型的视图。几乎每个项目都需要有一个页面来显示一个对象的列表（`List View`），或者一个对象的具体内容（`Detail View`），有或者是一个用来创建对象的表单。依照DRY精神，这些可重复使用的视图被Django打包在一起。

这里给出Django 1.7 中通用视图速查表格：

类型	类名称	描述

基本|*View*|这是所有视图的父类。它执行派遣和清醒检测。 基本|*TemplateView*|传递模板。暴露*URLConf*的关键字到上下文。 基本|*RedirectView*|对任意*GET*请求做出重定向。 列表|*ListView*|传递任意可迭代的项，比如*queryset*。 详细|*DetailView*|传递基于来自*URLConf*的*pk*或者*slug*。 编辑|*FormView*|传递并处理表单 编辑|*CreateView*|传递并处理生成新对象的表单。 编辑|*UpdateView*|传递并处理更新对象的表单。 编辑|*DeleteView*|传递并处理删除对象的表单。 日期|*ArchiveIndexView*|传递含有日期字段的对象列表，并把最新的日期放在最前面。 日期|*YearArchiveView*|传递基于*URLConf*中所给定的*year*的对象列表 日期|*MonthArchiveView*|传递基于*year*和*month*的对象列表。 日期|*WeekArchiveView*|传递基于*year*和*week*数的对象列表。 日期|*DayArchiveView*|传递基于*year*，*month*和*day*的对象列表。 日期|*TodayArchiveView*|传递基于当日日期的对象列表。 日期|*DateDetailView*|传递一个基于*year*，*month*，和*day*，通过自身的*pk*或者*slug*来区分的对象。

我们已经提过类似`BaseDetailView`这样的基类，或是`SingleObjectMixin`这样的mixins。它们设计成父类。多数情况下，你不会直接地运用它们。

大多数人对基于类的视图和基于类的通用视图感到迷惑。因为它们的名称相似，但是它们并非是同一个东西。这导致一些令人担心的误会：

```
*通用视图仅仅是Django的一组集合*：谢天谢地，这是错的。在基于类的通用视图中是没有什么特殊魔法的。
它们省去了你创建自己的一组基于类的通用视图的麻烦。你也可以使用类似`django-vanilla-views`这样的第三方库

*基于类的视图必须总是从一个通用视图中派生*：同样地，通用视图类也没有什么魔法。尽管，有百分之九十的时间，你
```

视图mixin

Mixin是在基于类的视图中按照DRY原则写代码的精髓所在。就像模型mixin一样，视图mixin得益于Python的多重继承，因此它可以很好的重复使用大部分的功能。在Python 3 中它们常常是无父类的类（或者是在Python 2 中新格式的类都派生自`object`）。

Mixin会在定义过的地方拦截视图的处理过程。例如，大多数的通用视图使用 `get_context_data` 设置上下文字典。它是一个插入额外上下文的好地方，比如一个用户可以查看所有指向发布文章的 `feed` 变量，一如下面命令所示：

```
class FeedMixin(object):
    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context["feed"] = models.Post.objects.viewable_posts(self.request.user)
        return context
```

首先 `get_context_data` 方法在基类中调用所有与自己同名的方法来生成上下文。接下来，他用 `feed` 变量更新上下文字典。

现在，在基类的列表中，这个mixin通过包含它就可以很容易地来添加用户的订阅。这就是说，如果SuperBook需要有一个发布新文章便可被订阅的表单的典型社交网络主页，你可以像下面这样使用这个mixin：

```
class MyFeed(FeedMixin, generic.CreateView):
    model = models.Post
    template_name = "myfeed.html"
    success_url = reverse_lazy("my_feed")
```

一个写的很好的mixin比需要非常高的要求。它应该在多数情况下都可以被灵活运用。在前面的例子中，`FeedMixin` 会重写派生类中的 `feed` 上下文变量。如果父类要把 `feed` 用作上下文变量，它可以通过包含这个mixin发挥作用。因此，在使用mixin之前，你需要检查mixin的源代码以及其他的类，以确保没有方法或者上下文变量的冲突。

mixins的顺序

你或许见过有多个mixins的代码：

```
class ComplexView(MyMixin, YourMixin, AccesMixin, DetailView):
```

It can get quite tricky to figure out the order to list the base classes. Like most things in Django, the normal rules of Python apply. Python's **Method Resolution Order** (MRO) determines how they should be arranged.

要找到列出基类的顺序是非常棘手的一件事。就像Django中的大多数情形一样，要用到就是Python的基本规则。Python的方法解析顺序决定了它们该如何被排列出来。

In a nutshell, mixins come first and base classes come last. The more specialied the parent class is, the more it moves to the left. In practice, this is the only rule you will need to remember.

简单来说就是，把mixin放在最前面，而基类放在最后面。如果有更多的父类，这些父类都会被放到左边。

To understand why this works, consider the following simple example:

```
class A:
       def do(self):
           print("A")

class B:
    def do(self):
           print("B")

class BA(B, A):
       pass

class AB(A, B):
       pass

BA().do() # Prints B
AB().do() # Prints A
```

As you would expect, if B is mentioned before A in the list of base classes, then B's method gets called and vice versa.

Now imagine A is a base class such as CreateView and B is a mixin such as FeedMixin. The mixin is an enhancement over the basic functionality of the base class. Hence, the ixin code should act first and in turn, call the base method if needed. So, the correct order is BA ixins first, base last.

The order in which base classes are called can be determined by checking the `__mro__` attribute of the class:

```
>>> AB.__mro__
 (__main__.AB, __main__.A, __main__.B, object)
```

So, if AB calls `super()`, first A gets called; then, A's `super()` will call B, and so on.

> `Python's MRO usually follows a depthfirst, lefttoright order to select a method in the c`

装饰器

Before class-based views, decorators were the only way to change the behavior of function-based views. Being wrappers around a function, they cannot change the inner working of the view, and thus effectively treat them as black boxes.

A decorator is function that takes a function and returns the decorated function. onfused? There is soe syntactic sugar to help you. se the annotation notation @, as shown in the following login_required decorator example:

```
@login_required
def simple_view(request):
        return HttpResponse()
```

The following code is exactly same as above:

```
def simple_view(request):
        return HttpResponse()
    simple_view = login_required(simple_view)
```

Since `login_required` wraps around the view, a wrapper function gets the control first. If the user was not logged in, then it redirects to `settings.LOGIN_URL`. Otherwise, it executes `simple_view` as if it did not exist. Decorators are less exible than ixins. However, they are sipler. You can use both decorators and mixins in Django. In fact, many mixins are implemented with decorators.

视图模式

Let's take a look at some common design patterns seen in designing views.

模式-访问控制视图

问题：页面需要基于用户是否登录，是否是站点成员，或者其他的任何条件，按照条件访问。

解决方法：使用mixins或者装饰器控制到视图的访问。

问题细节

大多数的网站都有当你登录才能访问的页面。另外的一些页面可以被匿名访问或者普通游客访问。如果一个匿名访客视图访问一个面向已登录用户的页面，它们会被路由到登录页面。理论上，在登录后，他们应该被路由回第一次想要见到的页面。

方案详情

有两个控制到一个视图的访问方法：

```
1. 通过对基于函数视图或者基于类视图使用一个装饰器实现控制：
```

```
@login_required(MyView.as_view())
```

```
2. 通过覆盖mixin的类视图的`dispatch`方法实现控制：
```

```
 class LoginRequiredMixin:
          @method_decorator(login_required)
          def dispatch(self, request, *args, **kwargs):
              return super().dispatch(request, *args, **kwargs)
```

```
这里我们真的不需要装饰器。更为明确地形式建议如下：
```

```
class LoginRequiredMixin:
           def dispatch(self, request, *args, **kwargs):
               if not request.user.is_authenticated():
                   raise PermissionDenied
               return super().dispatch(request, *args, **kwargs)
```

当异常 `PermissionDenied` 抛出时，Django会在根目录中显示 `403.html` 模板，如果模板不存在，就会出现“403 Forbidden”页面。

当然，

这里使用它们控制到登录和匿名访问的视图：

```
from braces.views import LoginRequiredMixin, AnonymousRequiredMixin
   class UserProfileView(LoginRequiredMixin, DetailView):
       # This view will be seen only if you are logged-in
       pass
   class LoginFormView(AnonymousRequiredMixin, FormView):
       # This view will NOT be seen if you are loggedin
       authenticated_redirect_url = "/feed"
```

Django中站点成员是使用设置在用户模型中的 `is_staff` 标识的用户。
同样地，你可以使用称做 `UserPassesTestMixin` 的django-braces mixin：

```
    from braces.views import UserPassesTestMixin

    class SomeStaffView(UserPassesTestMixin, TemplateView):
        def test_func(self, user):
            return user.is_staff
```

你也可以创建执行特定检查的mixins，比如对象是否被原作者或者其他人（使用登录用户比对）编辑：

```
    class CheckOwnerMixin:
        # 被用于派生自SingleObjectMixin的类
        def get_object(self, queryset=None):
            obj = super().get_object(queryset)
            if not obj.owner == self.request.user:
                raise PermissionDenied
            return obj
```

模式-上下文加强器

问题：多个基于类的通用视图需要相同的上下文变量。

解决方法：创建一个设置为共享上下文变量的mixin。

问题细节

Django templates can only show variables that are present in its context dictionary. However, sites need the same information in several pages. For instance, a sidebar showing the recent posts in your feed might be needed in several views.

However, if we use a generic class-based view, we would typically have a limited set of context variables related to a specific odel. etting the sae context variable in each view is not DRY.

方案详情

Most generic class-based views are derived from ContextMixin. It provides the get_context_data method, which most classes override, to add their own context variables. While overriding this method, as a best practice, you will need to call get_context_data of the superclass first and then add or override your context variables.

We can abstract this in the form of a mixin, as we have seen before:

```
class FeedMixin(object):
       def get_context_data(self, **kwargs):
           context = super().get_context_data(**kwargs)
           context["feed"] = models.Post.objects.viewable_posts(self.
   request.user)
           return context
```

We can add this mixin to our views and use the added context variables in our teplates. otice that we are using the odel anager defined in Chapter 3, Models, to filter the posts.

A more general solution is to use StaticContextMixin from django-braces for static-context variables. For example, we can add an additional context variable latest_profile that contains the latest user to join the site:

```
class CtxView(StaticContextMixin, generic.TemplateView):
       template_name = "ctx.html"
       static_context = {"latest_profile": Profile.objects.latest('pk')}
```

Here, static context means anything that is unchanged from a request to request. In that sense, you can mention QuerySets as well. However, our feed context variable needs self.request.user to retrieve the user's viewable posts. Hence, it cannot be included as a static context here.

模式-服务

问题：Information from your website is often scraped and processed by other applications.

解决方法：创建返回机器友好的格式的轻量的服务，比如JSON或者XML。

问题细节

We often forget that websites are not just used by huans. significant percentage of web traffic coes fro other progras like crawlers, bots, or scrapers. Sometimes, you will need to write such programs yourself to extract information from another website.

Generally, pages designed for human consumption are cumbersome for mechanical extraction. HTML pages have information surrounded by markup, requiring extensive cleanup. Sometimes, information will be scattered, needing extensive data collation and transformation.

 achine interface would be ideal in such situations. You can not only reduce the hassle of extracting information but also enable the creation of mashups. The longevity of an application would be greatly increased if its functionality is exposed in a machine-friendly manner.

方案详情

Service-oriented architecture (SOA) has popularized the concept of a service. A service is a distinct piece of functionality exposed to other applications as a service. For example, Twitter provides a service that returns the most recent public statuses.

A service has to follow certain basic principles:

```
• Statelessness: This avoids the internal state by externalizing state information
• Loosely coupled: This has fewer dependencies and a minimum of assumptions
• Composable: This should be easy to reuse and combine with other services
```

In Django, you can create a basic service without any third-party packages. Instead of returning HTML, you can return the serialized data in the JSON format. This form of a service is usually called a web Application Programming Interface (API).

For exaple, we can create a siple service that returns five recent public posts from SuperBook as follows:

```
class PublicPostJSONView(generic.View):
        def get(self, request, *args, **kwargs):
            msgs = models.Post.objects.public_posts().values(
                "posted_by_id", "message")[:5]
            return HttpResponse(list(msgs), content_type="application/json")
```

For a more reusable implementation, you can use the `JSONResponseMixin` class from `django-braces` to return JSON using its `render_json_response` method:

```
from braces.views import JSONResponseMixin
   class PublicPostJSONView(JSONResponseMixin, generic.View):
       def get(self, request, *args, **kwargs):
           msgs = models.Post.objects.public_posts().values("posted_by_id", "message")[:5
           return self.render_json_response(list(msgs))
```

If we try to retrieve this view, we will get a JSON string rather than an HTML response:

```
>>> from django.test import Client
>>> Client().get("http://0.0.0.0:8000/public/").content
b'[{"posted_by_id": 23, "message": "Hello!"},
   {"posted_by_id": 13, "message": "Feeling happy"},
   ...
```

Note that we cannot pass the QuerySet method directly to render the JSON response. It has to be a list, dictionary, or any other basic Python built-in data type recognized by the JSON serializer.

Of course, you will need to use a package such as Django REST framework if you need to build anything more complex than this simple API. Django REST framework takes care of serializing (and deserializing) QuerySets, authentication, generating a web-browsable API, and many other features essential to create a robust and fulledged PI.

设计URL

Django has one of the ost exible RL schees aong web fraeworks. Basically, there is no iplied RL schee. You can explicitly define any RL scheme you like using appropriate regular expressions.

However, as superheroes love to say—"With great power comes great responsibility. You cannot get away with a sloppy RL design any more.

URLs used to be ugly because they were considered to be ignored by users. Back in the 90s when portals used to be popular, the common assumption was that your users will come through the front door, that is, the home page. They will navigate to the other pages of the site by clicking on links.

Search engines have changed all that. According to a 2013 research report, nearly half (47 percent) of all visits originate from a search engine. This means that any page in your website, depending on the search relevance and popularity can be the first page your user sees. ny RL can be the front door.

More importantly, Browsing 101 taught us security. Don't click on a blue link in the wild, we warn beginners. Read the RL first. Is it really your bank's RL or a site trying to phish your login details?

Today, URLs have become part of the user interface. They are seen, copied, shared, and even edited. Make them look good and understandable from a glance. No more eye sores such as:

```
http://example.com/gallery/default.asp?sid=9DF4BC0280DF12D3ACB6009027
1E26A8&command=commntform
```

Short and meaningful URLs are not only appreciated by users but also by search engines. URLs that are long and have less relevance to the content adversely affect your site's search engine rankings.

Finally, as implied by the maxim "Cool URIs don't change," you should try to maintain your URL structure over time. Even if your website is completely redesigned, your old links should still work. Django makes it easy to ensure that this is so.

Before we delve into the details of designing URLs, we need to understand the structure of a URL.

URL解剖

Technically, URLs belong to a ore general faily of identifiers called **Uniform Resource Identifiers (URIs)**. Hence, a URL has the same structure as a URI.

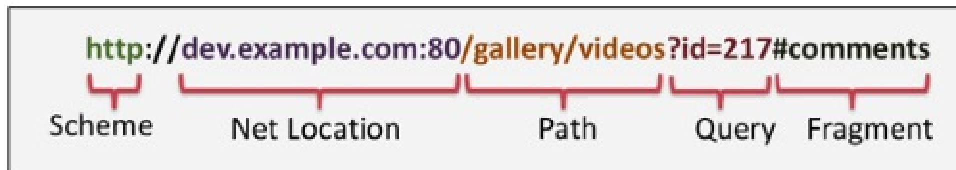
A URI is composed of several parts:

```
URI = Scheme + Net Location + Path + Query + Fragment
```

For example, a URI (http://dev.example.com:80/gallery/ videos?id=217#comments) can be deconstructed in Python using the urlparse function:

```
>>> from urllib.parse import urlparse
>>> urlparse("http://dev.example.com:80/gallery/videos?id=217#comments")
ParseResult(scheme='http', netloc='dev.example.com:80', path='/gallery/
videos', params='', query='id=217', fragment='comments')
```

The URI parts can be depicted graphically as follows:



Even though Django documentation prefers to use the term URLs, it might more technically correct to say that you are working with URIs most of the time. We will use the terms interchangeably in this book.

Django URL patterns are mostly concerned about the 'Path' part of the URI. All other parts are tucked away.

url.py中发生了什么？

It is often helpful to consider urls.py as the entry point of your project. It is usually the first file I open when I study a Django project. ssentially, urls.py contains the root RL configuration or URLConf of the entire project.

It would be a Python list returned from patterns assigned to a global variable called urlpatterns. Each incoming URL is matched with each pattern from top to botto in a sequence. In the first atch, the search stops, and the request is sent to the corresponding view.

Here, in considerably siplified for, is an excerpt of urls.py from Python.org, which was recently rewritten in Django:

```
urlpatterns = patterns(
        '',
        # Homepage
        url(r'^$', views.IndexView.as_view(), name='home'),
        # About
        url(r'^about/$',
            TemplateView.as_view(template_name="python/about.html"),
            name='about'),
        # Blog URLs
        url(r'^blogs/', include('blogs.urls', namespace='blog')),
        # Job archive
        url(r'^jobs/(?P<pk>\d+)/$',
            views.JobArchive.as_view(),
            name='job_archive'),
        # Admin
        url(r'^admin/', include(admin.site.urls)),
    )
```

Some interesting things to note here are as follows:

```
• The first arguent of the patterns function is the prefix. It is usually blank
for the root URLConf. The remaining arguments are all URL patterns.
• Each URL pattern is created using the url function, which takes five arguments. Most pa
• The about pattern defines the view by directly instantiating TemplateView. Some hate th
• Blog RLs are entioned elsewhere, specifically in urls.py inside the blogs app. In gener
• The jobs pattern is the only example here of a named regular expression.
```

In future versions of Django, urlpatterns should be a plain list of URL pattern objects rather than arguments to the patterns function. This is great for sites with lots of patterns, since urlpatterns being a function can accept only a maximum of 255 arguments.

If you are new to Python regular expressions, you ight find the pattern syntax to be slightly cryptic. Let's try to demystify it.

URL模式语法

URL regular expression patterns can sometimes look like a confusing mass of punctuation marks. However, like most things in Django, it is just regular Python.

It can be easily understood by knowing that URL patterns serve two functions: to match URLs appearing in a certain form, and to extract the interesting bits from a URL.

The first part is easy. If you need to atch a path such as `/jobs/1234` , then just use the `"^jobs/\d+"` pattern (here `\d` stands for a single digit from `0` to `9`). Ignore the leading slash, as it gets eaten up. The second part is interesting because, in our example, there are two ways of extracting the job ID (that is, `1234`), which is required by the view.

The simplest way is to put a parenthesis around every group of values to be captured. Each of the values will be passed as a positional argument to the view. For example, the `"^jobs/(\d+)"` pattern will send the value "1234" as the second arguent the first being the request to the view.

The problem with positional arguments is that it is very easy to mix up the order. Hence, we have name-based arguments, where each captured value can be named. Our example will now look like `"^jobs/(?P<pk>\d+)/"` . This means that the view will be called with a keyword argument pk being equal to "1234".

If you have a class-based view, you can access your positional arguments in `self. args` and name-based arguments in `self.kwargs` . Many generic views expect their arguments solely as name-based arguments, for example, self.kwargs["slug"].

Mnemonic – parents question pink action-figures

I admit that the syntax for name-based arguments is quite difficult to remember. Often, I use a simple mnemonic as a memory aid. The phrase "Parents Question Pink Action-figures" stands for the first letters of Parenthesis, Question mark, (the letter) P, and Angle brackets.

Put them together and you get (`?P<` . You can enter the name of the pattern and figure out the rest yourself.

It is a handy trick and really easy to remember. Just imagine a furious parent holding a pink-colored hulk action figure.

Another tip is to use an online regular expression generator such as http://pythex. org/ or https://www.debuggex.com/tocraftandtestyourregularexpressions.

命名和命名空间

Always name your patterns. It helps in decoupling your code from the exact URL paths. For instance, in the previous URLConf, if you want to redirect to the about page, it might be tempting to use redirect("/about"). Instead, use redirect("about"), as it uses the name rather than the path. Here are some more examples of reverse lookups:

```
>>> from django.core.urlresolvers import reverse
>>> print(reverse("home"))
"/"
>>> print(reverse("job_archive", kwargs={"pk":"1234"}))
"jobs/1234/"
```

Names must be unique. If two patterns have the same name, they will not work. So, some Django packages used to add prefixes to the pattern name. For example, an application named blog might have to call its edit view as 'blog-edit' since 'edit' is a common name and might cause conflict with another application.

Namespaces were created to solve such problems. Pattern names used in a namespace have to be only unique within that namespace and not the entire project. It is recommended that you give every app its own namespace. For example, we can create a 'blog' namespace with only the blog's URLs by including this line in the root URLconf:

```
    url(r'^blog/', include('blog.urls', namespace='blog')),
```

Now the blog app can use pattern names, such as 'edit' or anything else as long as they are unique within that app. While referring to a name within a namespace, you will need to mention the namespace, followed by a ':' before the name. It would be `"blog:edit"` in our example.

As Zen of Python says—"Namespaces are one honking great idea—let's do more of those." You can create nested namespaces if it makes your pattern names cleaner, such as "blog:comment:edit". I highly recommend that you use namespaces in your projects.

模式顺序

Order your patterns to take advantage of how Django processes them, that is, top-down. A good rule of thumb is to keep all the special cases at the top. Broader patterns can be mentioned further down. The broadest—a catch-all—if present, can go at the very end.
For example, the path to your blog posts might be any valid set of characters, but you might want to handle the About page separately. The right sequence of patterns should be as follows:

```
urlpatterns = patterns(
    '',
    url(r'^about/$', AboutView.as_view(), name='about'),
    url(r'^(?P<slug>\w+)/$', ArticleView.as_view(), name='article'),
)
```

If we reverse the order, then the special case, the AboutView, will never get called.

URL模式风格

Designing URLs of a site consistently can be easily overlooked. Well-designed URLs can not only logically organize your site but also make it easy for users to guess paths. Poorly designed ones can even be a security risk: say, using a database ID (which occurs in a monotonic increasing sequence of integers) in a URL pattern can increase the risk of information theft or site ripping.

Let's examine some common styles followed in designing URLs.

分布存储URL

Some sites are laid out like Departmental stores. There is a section for Food, inside which there would be an aisle for Fruits, within which a section with different varieties of Apples would be arranged together.
In the case of URLs, this means that you will find these pages arranged hierarchically as follows:

```
http://site.com/ <section> / <sub-section> / <item>
```

The beauty of this layout is that it is so easy to climb up to the parent section. Once you remove the tail end after the slash, you are one level up.

For example, you can create a similar structure for the articles section, as shown here:

```
    # project's main urls.py
    urlpatterns = patterns(
'',
        url(r'^articles/$', include(articles.urls), namespace="articles"),
    )
    # articles/urls.py
    urlpatterns = patterns(
        '',
        url(r'^$', ArticlesIndex.as_view(), name='index'),
        url(r'^(?P<slug>\w+)/$', ArticleView.as_view(), name='article'),
)
```

Notice the 'index' pattern that will show an article index in case a user climbs up from a particular article.

RESTful URLs

In 2000, Roy Fielding introduced the term Representational state transfer (REST) in his doctoral dissertation. Reading his thesis (http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm) is highly recommended to better understand the architecture of the web itself. It can help you write better web applications that do not violate the core constraints of the architecture.

One of the key insights is that a URI is an identifier to a resource. A resource can be anything, such as an article, a user, or a collection of resources, such as events. Generally speaking, resources are nouns.

The web provides you with some fundamental HTTP verbs to manipulate resources: GET, POST, PUT, PATCH, and DELETE. Note that these are not part of the URL itself. Hence, if you use a verb in the URL to manipulate a resource, it is a bad practice.

For example, the following URL is considered bad:

```
http://site.com/articles/submit/
```

Instead, you should remove the verb and use the POST action to this URL:

```
http://site.com/articles/
```

> ## Best Practice
>
> Keep verbs out of your URLs if HTTP verbs can be used instead.

Note that it is not wrong to use verbs in a URL. The search URL for your site can have the verb 'search' as follows, since it is not associated with one resource as per REST:

```
http://site.com/search/?q=needle
```

RESTful URLs are very useful for designing CRUD interfaces. There is almost a one-to-one mapping between the Create, Read, Update, and Delete database operations and the HTTP verbs.

Note that the RESTful URL style is complimentary to the departmental store URL style. Most sites mix both the styles. They are separated for clarity and better understanding.

下载练习代码

You can download the example code fies for all Packt books you have purchasedfrom your account at http://www.packtpub.com. If you purchased this bookelsewhere, you can visit http://www.packtpub. com/support and register tohave the fies e-mailed directly to you. Pull requests and bug reports to the SuperBook project can be sent to https://github.com/DjangoPatternsBook/superbook.

总结

Views are an extremely powerful part of the MVC architecture in Django. Over time, class-based views have proven to be more flexible and reusable compared to traditional function-based views. Mixins are the best examples of this reusability.

Django has an extremely flexible URL dispatch system. Crafting good URLs takes into account several aspects. Well-designed URLs are appreciated by users too. In the next chapter, we will take a look at Django's templating language and how best to leverage it.

第五章 模板

本章，我们会讨论以下议题：

```
1. Django模板语法的特性
2. 组织模板
3. Bootstrap
4. 模板继承树模式
5. 活动连接模式
```

理解Django的模板语言特点

是时候谈谈MTV模板三件套中的第三个东西了。而你的团队或许有关心模板设计的设计者。或者你想要自己设计模板。无论选择哪种方式，你都需要对它们非常熟悉。毕竟，模板是直接面向用户的。

我们对Django模板语言特性进行快速入门。

变量

每个模板都获取一组上下文变量。类似于Python的字符串 `format()` 方法的单大括号 `{variable}` 语法，Django使用双大括号 `{{ variable }}` 语法。我们来看看它们之间的比较：

- 在纯净的Python中语法为`<h1>{title}</h1>`。例如：

```
>>> "<h1>{title}</h1>".format(title="SuperBook") '<h1>SuperBook</h1>'
```

- Django模板中的相等的语法是 `<h1>{{ title }}</h1>`.
- 如下传递下相同的上下文产生同样的结果：

```
>>> from django.template import Template, Context
>>> Template("<h1>{{ title }}</h1>").render(Context({"title":
"SuperBook"}))
'<h1>SuperBook</h1>'
```

属性

点号在Django模板中是多用途的运算符。这里有三种不同类型的运算符，属性查找，字典查找，列表索引查找（依照顺序）。

```
• Python中我们首先，定义上下文变量和类：
```

```
>>> class DrOct:
        arms = 4
        def speak(self):
            return "You have a train to catch."
>>> mydict = {"key":"value"}
>>> mylist = [10, 20, 30]
```

```
我们来看看三种查询类型的Python语法：
```

```
>>> "Dr. Oct has {0} arms and says: {1}".format(DrOct().arms,
DrOct().speak())
'Dr. Oct has 4 arms and says: You have a train to catch.'
>>> mydict["key"]
 'value'
>>> mylist[1]
 20
```

```
• Django的模板等于下面:
```

```
Dr. Oct has {{ s.arms }} arms and says: {{ s.speak }}
{{ mydict.key }}
{{ mylist.1 }}
```

> 注释
>
> 注意*speak*方法没有接受参数，除了被当作属性的*self*之外。

过滤器

某些时候，变量是需要修改的。从根本上来说，你要利用这些变量去调用函数。Django使用类似于Unix过滤器的管道语法 `{{ var|method1|method2:"tag" }}` ，而不是 `var.method().method2(arg)` 这样的链式函数调用。

过滤器的另外一个限制是它不能够访问模板上下文。过滤器仅在数据传递到过滤器，以及过滤器的参数时才有效。因此，在模板上下文中它主要用来更改变量。

- 在Python中运行以下命令:

```
>>> title="SuperBook"
>>> title.upper()[:5]
'SUPER'
```

- 其对应的Django模板为:

```
{{ title|upper|slice:':5' }}
```

标签

编程语言能够做的事情不仅仅是显示变量。Django的模板语言多中相似的语法形式，比如 `if` 和 `for` 。它们应该用 `{% if %}` 这样的语法写成。多个针对模板的形式，比如 `include` 和 `block` 也都是用标签语法写成的。

```
• 在Python中运行以下命令:
```

```
>>> if 1==1:
...     print(" Date is {0} ".format(time.strftime("%d-%m-%Y")))
 Date is 31-08-2014
```

```
• 其对应的Django模板形式如下:
```

```
{% if 1 == 1 %} Date is {% now 'd-m-Y' %} {% endif %}
```

哲学——不要去发明一种编程语言

新手们间的常见问题是如何执行发现模板中百分比这样的数字计算。从设计哲学的角度来说，模板系统无意支持这样做：

- 对变量赋值
- 高级逻辑

该决定能够阻止你在模板中添加业务逻辑。根据使用PHP或者类ASP语言的经历，将逻辑和表现层混合到一起是一场后期维护的噩梦。不过，你可以编写自定义的模板标签（很快就会学到）以执行任何计算，特别是和表现层相关的计算。

> 提示
>
> 最佳实践 保证业务逻辑远离模板。

组织模板

由startproject命令创建的默认项目布局并没有定义好的模板的位置。这个问题解决起来也非常简单。在项目的根目录中创建一个名称为 `templates` 的目录。然后在 `settings.py` 目录中添加 `TEMPLATE_DIRS` 变量。

> 译者注 在Django 1.8之后的版本中，已经不用手动定义模板路径。startproject命令创建的默认配置文件已经自动包含了项目根目录下的templates目录。
>
> ```
> TEMPLATES = [
>     {
>         'BACKEND': 'django.template.backends.django.DjangoTemplates',
>         'DIRS': [os.path.join(BASE_DIR, 'templates')],
>         'APP_DIRS': True,
>         'OPTIONS': {
>             'context_processors': [
>                 'django.template.context_processors.debug',
>                 'django.template.context_processors.request',
>                 'django.contrib.auth.context_processors.auth',
>                 'django.contrib.messages.context_processors.messages',
>             ],
>         },
>     },
> ]
> ```

```
BASE_DIR = os.path.dirname(os.path.dirname(__file__))
TEMPLATE_DIRS = [os.path.join(BASE_DIR, 'templates')]
```

```
要定义的就这么多。例如，你可以添加一个称做`about.html`的模板，然后在`urls.py`文件中引用它：

```python
urlpatterns = patterns('',url(r'^about/$', TemplateView.as_view(template_name='about.html
```

模板也可以定义在应用中。在应用的目录中创建模板目录对存储应用专用的模板来说是非常理想的。

下面是组织模板的一些优秀实践：

- 再一个单独的目录中，保证所有app专用的模板都放在这个app的模板目录汇总，例如， `projroot/app/templates/app/template` 。
- 对模板使用.html这样对扩展名。
- 对模板添加一个下划线前缀，表示将要继承使用对代码片段，例如， `_navbar.html` 。— 将应用专用的模板放到一个独立的应用的模板目录中，例如， `projecroot/app/templates/app/template` ，而 `html` 也能够理解 `app` 两次出现在路径的原因。
- 为模板使用 `.html` 格式的文件扩展名。
- 对模板使用带下划线的前缀，该模板是能够在继承中使用的代码片段，例如， `_navbar.html` 。

对其他模板语言的支持

从Django1.8起多模板引擎将得到支持。届时会存在内建的Django模板语言（我们之前谈到的普通模板语言）和Jinja2的支持。在很多的性能基准测试中，Jinja2比Django模板快了很多。
```

对于指定的模板引擎和所有模板相关的设置来说都需要存在额外的TEMPLATES设置。而TEMPLATE_DIRS设置很快也会被移除。

欧女士

For the first time in weeks, Steve's office corner was bustling with frenetic activity. With more recruits, the now five-member team comprised of Brad, Evan, Jacob, Sue, and Steve. Like a superhero team, their abilities were deep and amazingly well-balanced.

Brad and Evan were the coding gurus. While Evan was obsessed over details, Brad was the big-picture guy. Jacob's talent in finding corner cases made him perfect for testing. Sue was in charge of marketing and design.

In fact, the entire design was supposed to be done by an avant-garde design agency. It took them a month to produce an abstract, vivid, color-splashed concept loved by the management. It took them another two weeks to produce an HTML-ready version from their Photoshop mockups. However, it was eventually discarded as it proved to be sluggish and awkward on mobile devices.

Disappointed by the failure of what was now widely dubbed as the "unicorn vomit" design, Steve felt stuck. Hart had phoned him quite concerned about the lack of any visible progress to show management. In a grim tone, he reminded Steve, "We have already eaten up the project's buffer time. We cannot afford any last-minute surprises."

It was then that Sue, who had been unusually quiet since she joined, mentioned that she had been working on a mockup using Twitter's Bootstrap. Sue was the growth hacker in the team—a keen coder and a creative marketer.

She admitted having just rudimentary HTML skills. However, her mockup was surprisingly thorough and looked familiar to users of other contemporary social networks. Most importantly, it was responsive and worked perfectly on every device from tablets to mobiles.

The management unanimously agreed on Sue's design, except for someone named Madame O. One Friday afternoon, she stormed into Sue's cabin and began questioning everything from the background color to the size of the mouse cursor. Sue tried to explain to her with surprising poise and calm.

An hour later, when Steve decided to intervene, Madame O was arguing why the profile pictures must be in a circle rather than square. "But a site-wide change like that will never get over in time," he said. Madame O shifted her gaze to him and gave him a sly smile. Suddenly, Steve felt a wave of happiness and hope surge within him. It felt immensely reliving and stimulating. He heard himself happily agreeing to all she wanted.

Later, Steve learnt that Madame Optimism was a minor mentalist who could influence prone minds. His team loved to bring up the latter fact on the slightest occasion.

使用Bootstrap

对任何人来说开始从零建设整个网站的那些日子都是件很艰苦的事情。像推特的或者说是扎克伯格基金会的Bootstrap这样的CSS框架，栅格系统，极好的排列板式以及预先设置好的风格，下手好起点。使用Bootstrap的大多数网页设计都对移动设备显示友好。

图片：略

我们将使用Bootstrap，而且步骤也类似于其他的CSS框架。在网站中使用Bootstrap有三种方法：

- Find a project skeleton: If you have not yet started your project, then finding a project skeleton that already has Bootstrap is a great option. A project skeleton such as edge (created by yours truly) can be used as the initial structure while running startproject as follows:
- 寻找项目架构：如果还没有启动项目，那么查找项目架构

```
$ django-admin.py startproject --template=https://github.com/arocks/edge/archive/master.z
```

可选择的是，你可以使用其中一个拥有Bootstrap支持的cookiecutter模板。

- 使用包：最简单的选项是如果你已经开始对自己的项目使用包，比如django-frontend-skeleton 或者 django-bootstrap-toolkit。
- 手动复制：之前的选项并不能保证Bootstrap的版本是最新的。而Bootstrap发布的很频繁，所以包的捉着很难保证其中文件的更新。因此，如果你想要使用最新版本的Bootstrap，最好的选择是自己到这个地址下载 http://getbootstrap.com。要记得去阅读发型注释，以便检查你的模版是否需要为了向后兼容行而作出改变。

复制包含css、js的dist目录，以及静态目录下根目录中的字体目录。保证这个路径设置在settings.py中的STATICFILES_DIRS：

```
      STATICFILES_DIRS = [os.path.join(BASE_DIR, "static")]
   Now you can include the Bootstrap assets in your templates, as follows:
      {% load staticfiles %}
        <head>
          <link href="{% static 'css/bootstrap.min.css' %}"
      rel="stylesheet">
```

怎么它们看上去都一个模样啊！

Bootstrap或许是快速开发的非常好的选择。不过，有时候，开发者懒得去改变默认的外观。这就留给了浏览网站的用户一个糟糕的映像，他们发现你的网站的外观也太熟悉了而且很无趣。

Bootstrap带来了很多的改进外观需求的选项。如下，有一个称作 `variables.less` 的文件包含了来自默认字体的主要种类色彩中的多个变量：

```
@brand-primary: #428bca;
@brand-success: #5cb85c;
@brand-info: #5bc0de;
@brand-warning: #f0ad4e;
@brand-danger: #d9534f;
@font-family-sans-serif:  "Helvetica Neue", Helvetica, Arial, sans-
serif;
@font-family-serif:
@font-family-monospace:
monospace;
@font-family-base:
Georgia, "Times New Roman", Times, serif;
Menlo, Monaco, Consolas, "Courier New",
@font-family-sans-serif;
```

Bootstrap文档解释了你该如何设置构造系统（包含LESS编译器在内），以便讲这些文件编译为样式表。另外一个很方便的选择是你去访问Bootstrap往后在哪的“定制”区域以在线生成定制的样式表。

要感谢大量的有关Bootstrap的社区，这里有很多网站，比如bootswatch.com，该站点拥有可以替换bootstrap.min.css的主题化样式表。

另外一个方法是重写Bootstrap版式。如果发现再不同版本的Bootstrap之间升级自定义的Bootstrap样式是多么无聊的事情，因为这只是个建议而已。在这个方法中，你可以在独立的CSS（或者LESS）文件中添加自己的全站样式，然后在标准的Bootstrap样式表中继承它。这样，你就能够以对全站样式表最小改变来简单的升级Bootstrap文件。

最后然后却不是最少的，你可以让自己CSS类更为有意义以替换结构式的名称，比如“row”或者'column-md-4', 和 'wrapper' 或者 'sidebar'。如下，你可以用几行LESS代码完成它：

```
.wrapper {
  .make-row();
 }
 .sidebar {
  .make-md-column(4);
}
```

这是处理一个功能的可行称作mixin的方法（听着很耳熟，是吧！）。使用LESS源码你可以完全依照自身需求完成定制。

模板模式

Django模板语言非常的简单。然而，它通过一些简洁的模板设计模式可以让你节省很多的时间。让我们来看看其中的一些模板设计模式。

模式-模板继承树

问题：在多个页面中模板存在很多重复的内容。

解决方案： 只要是有可能的地方就使用模板继承，并在其他地方继承代码片段。

问题细节

用户期望网站的页面能够遵循结构的一致性。某些接口元素，比如在很多web应用都能够见到的导航栏菜单，首部，和页脚。不过，在每一个模板中都重复它们显示相当笨重。

大多数的模板语言都拥有模板继承机制。其他文件的内容，也可以是一个模板都能够在它们被调用的地方导入。在大型项目中这样做会让你感到乏味的。

在每一个模板中将要继承的代码片段序列很大程度上是相同的。倒入顺序很重要，而错误检查却很难。理论上，我们能够创建一个`基础`结构。新的页面应当扩展这个基础模板

方案详情

Django模板拥有一个强大的扩展机制。类似于编程中的类，模板可以通过继承来扩展。不过，要让模板工作起来，base自身必要像下面这样组织到block中。

图片：略

The base.html template is, by convention, the base structure for the entire site. This template will usually be well-formed HTML (that is, with a preamble and matching closing tags) that has several placeholders marked with the {% block tags %} tag. For example, a minimal base.html file looks like the following:

我们约定，base.html 模板是真个网站的基本结构。这个模板

```
<html>
<body>
<h1>{% block heading %}Untitled{% endblock %}</h1>
{% block content %}
{% endblock %}
</body>
</html>
```

这里存在两个能够被重写块，heading和content。你可以扩展基础模板以创建能够重写这些块的特有页面。例如，这里是一个about页面：

```
{% extends "base.html" %}
{% block content %}
<p> This is a simple About page </p>
{% endblock %}
{% block heading %}About{% endblock %}
```

注意，我们不需要出现重复的结构。我们按照任意顺次饮用block。渲染结果将和定义在base.html的内容一样在正确的地方使用正确的block。

如果继承的模板没有重写block，那么它的父模板内容可以被使用。在前面的例子中，假如about模板不存在头部，那么它将拥有默认的'Untitled'头部。

The inheriting template can be further inherited forming an inheritance chain. This pattern can be used to create a common derived base for pages with a certain layout, for example, single-column layout. A common base template can also be created for a section of the site, for example, blog pages.

通常，所有的继承链都可以通过一个公共的根，base.html来回溯；因此，样式的名称——模板继承树。当然，这样的做法不需要到什么技巧的。而错误页面404.html和500.html通常并不需要继承，并去掉了大多数的标签以阻止更多错误的发生。

模式-活动链接

问题：在很多对页面中导航栏是一个常见对组件。然而，活动链接需要在用户登陆时进行响应的映射。

解决方案：根据情况不同，通过设置上下文变量或者请求路径来改变活动链接的外观。

问题细节

在导航栏内实现活动链接的简单办法是手动地在每个一个需要实现该功能的页面中设置。不过，这样过既不符合DRY原则，也并不能保证万无一失。

方案详情

确定活动链接的解决方法有多种。包括，基于Javascript的方法，它们主要是将纯模板和自定义标签组合到一起。

临时模板方案

在继承导航模板的代码片段时可以饮用一个称作active_link的变量，这个方案实现起来既简单又容易。

在每一个模板中，你都需要包括以下内容（或者继承它）：

```
{% include "_navbar.html" with active_link='link2' %}
```

`_navbar.html` 文件包含了拥有一组检查活动链接的变量的导航菜单：

```
 {# _navbar.html #}
<ul class="nav nav-pills">
  <li{% if active_link == "link1" %} class="active"{% endif %}><a
href="{% url 'link1' %}">Link 1</a></li>
  <li{% if active_link == "link2" %} class="active"{% endif %}><a
href="{% url 'link2' %}">Link 2</a></li>
  <li{% if active_link == "link3" %} class="active"{% endif %}><a
href="{% url 'link3' %}">Link 3</a></li>
</ul>
```

自定义标签

Django模板提供了一组多功能的内奸标签。创建自己自定义标签很简单。因此自定义标签是使用在应用内部的，那么就需要在应用的内部创建一个称作templatetags到目录。这个目录必须是一个Python包，所以应该包含一个（空的） `__init__.py` 文件。

接下来，在给定名称的Python文件中编写自己的自定义模板。例如，对于激活链接样式，我们可以使用一下内容创建一个称作nav.py的文件：

```
# app/templatetags/nav.py
from django.core.urlresolvers import resolve
from django.template import Library

register = Library()

@register.simple_tag
def active_nav(request, url):
   url_name = resolve(request.path).url_name
   if url_name == url:
       return "active"
   return ""
```

这个文件定义了一个称作 `active_nav` 的自定义标签。它从request参数重新取回URL的路径组件（即，第四章——视图与路由，其中有对URL路径对详细说明）。然后resolve()函数用来从路径查询URL的样式名称（定义在urls.py中）。最后，让仅在样式名称匹配期望的样式名称时反悔字符串“active”。

在模板中调用这个自定义标签的语法是 `{% active_nav request 'pattern_name' %}` 。注意，在用到这个标签的每个地方都需要将request传递进去。

在多个视图中使用一个变量显得太笨拙了。如下，我们在settings.py中添加一个内建的上下文处理器到 `TEMPLATE_CONTEXT_PROCESSORS` ,request便可以用request变量的身份出现在整个网站了：

```
# settings.py
from django.conf import global_settings

TEMPLATE_CONTEXT_PROCESSORS = \
    global_settings.TEMPLATE_CONTEXT_PROCESSORS + (
        'django.core.context_processors.request',
    )
```

现在，在模板中剩下的内容要使用这个定制标签，这样就可以设置激活属性了：

```
{# base.html #}
{% load nav %}
<ul class="nav nav-pills">
  <li class={% active_nav request 'active1' %}><a href="{% url
'active1' %}">Active 1</a></li>
  <li class={% active_nav request 'active2' %}><a href="{% url
'active2' %}">Active 2</a></li>
  <li class={% active_nav request 'active3' %}><a href="{% url
'active3' %}">Active 3</a></li>
</ul>
```

总结

在这一章，我们浏览了Django的模板特性。因为在Django中切换模板语言很简单，所以很多人都考虑替换。不过，在浏览其他的选择之前重要的是学些内建模板语言的设计哲学。

下一章，我们会学习Django的杀手级功能，即，admin接口，以及我们如何定制它。

第六章-Admin接口

这一章，我们会讨论以下话题：

- 定制admin
- 增强admin的模型
- Admin的最佳实践
- 特性标识

Django被谈论到最多的是，与其他的竞争对手相比它将admin接口独立了出来。。admin接口是一个自动地生成添加和修改一个站点内容的用户接口。不仅如此，admin也是Django的杀手级应用，它使项目中对模型生成admin接口的乏味的任务可以自动化。

admin能够让你的团队在同一时间内添加内容，不间断开发。只要模型已经应用了迁移，你仅需添加一行或者两行代码就可以生成模型的的admin接口。

使用admin接口

在Django1.7中，admin接口默认是启用的。在创建项目之后，你浏览 `http://127.0.0.1:800/admin` 时能够看到一个登录页面。

如果输入超级用户凭证（或者任意站点注册用户的凭证），那么你会登录到admin中，一如下面截图所示：

不过，模型在admin管理界面是不可见的，除非你定义一个与之对应的 `ModelAdmin` 类。这个操作通常定义在应用的admin.py文件，一如下面所示：

```
from django.contrib import admin
from . import models
admin.site.register(models.SuperHero)
```

此处，`ModelAdmin` 的到注册器的第二个参数被省略。因此，我们会获得一个默认的Post模型的admin接口。让我们看看如何创建并自定义 ``ModelAdmin` 类。

> ## 注释
>
> 引路人“还有咖啡吗☕？“一个声音来自备餐室角落的声音问道。苏差点儿把咖啡洒了出来。她前面站着一位身着紧身红蓝相间衣服 ，面带微笑，将手叉在腰间的高个子男人。
>
> "Oh, my god," said Sue as she wiped the coffee stain with a napkin. "Sorry, I think I scared you," said Captain Obvious "What is the emergency?" "Isn't it obvious that she doesn't know?" said a calm feminine voice from above. Sue looked up to find a shadowy figure slowly descend from the open hall. Her face was partially obscured by her dark matted hair that had a few grey streaks. "Hi Hexa!" said the Captain "But then, what was the message on SuperBook about?"
>
> “哎呦喂，”苏说道，同时另一边她小毛巾擦掉了泼出去的咖啡。“不好意啊，吓到你了，”装傻队长问道。“什么事这么急啊？”
>
> Soon, they were all at Steve's office staring at his screen. "See, I told you there is no beacon on the front page," said Evan. "We are still developing that feature." "Wait," said Steve. "Let me login through a non-staff account." In a few seconds, the page refreshed and an animated red beacon prominently appeared at the top. "That's the beacon I was talking about!" exclaimed Captain Obvious. "Hang on a minute," said Steve. He pulled up the source files for the new features deployed earlier that day. A glance at the beacon feature branch code made it clear what went wrong:
>
> 几秒钟之后，页面重新刷新，然后一个重要的红色提醒出现在顶部。“这就是我提到的信标！”装傻队长说道。“稍等一下啊，”斯蒂夫说到。他从前些天部署的新功能拉取了原文件。稍微看下信标功能的分支代码能够搞清楚到底哪里出问题了：

>

```
    if switch_is_active(request, 'beacon') and not
      request.user.is_staff():
           # Display the beacon
```

> “各位不好意思，”斯蒂夫说道。“这里出现了一个逻辑错误。我们一时疏忽就将该功能的启用开放给了所有人，而不是仅有站点注册用户来开启。现在我把这个功能给关闭了。对于因此而引起的混乱我表示歉意。”
>
> "So, there was no emergency?" said Captain with a disappointed look. Hexa put an arm on his shoulder and said "I am afraid not, Captain." Suddenly, there was a loud crash and everyone ran to the hallway. A man had apparently landed in the office through one of the floor-to-ceiling glass walls. Shaking off shards of broken glass, he stood up. "Sorry, I came as fast as I could," he said, "Am I late to the party?" Hexa laughed. "No, Blitz. Been waiting for you to join," she said.
>
> “这么说，根本没什么要紧事喽？”队长带着一脸失望☹的说道。Hexa一面把手搭在他肩膀上一边说道：“我也不希望发生什么事情，队长。”突然之间，室内闪现出一团云，屋里的人都赶紧跑开了。

增强用于admin的模型

admin应用足够聪明，因此它可以自动地从模型发现非常多东西。可是，有时候推定信息需要改进。这通常涉及到模型自身添加一个属性或者一个方法（而不是在 `ModelAdmin` 类中添加）。

首先，为了更好的说明问题，让我们来看一个包括admin接口而增强模型的例子：

```
# models.py
class SuperHero(models.Model):
    name = models.CharField(max_length=100)
    added_on = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return "{0} - {1:%Y-%m-%d %H:%M:%S}".format(self.name,
                                                     self.added_on)
    def get_absolute_url(self):
        return reverse('superhero.views.details', args=[self.id])

    class Meta:
        ordering = ["-added_on"]
        verbose_name = "superhero"
        verbose_name_plural = "superheroes"
```

我们看看admin是如何利用这些非字段属性的：

- __str__(): 没有它的话，superhero条目的列表看上去会极其无趣的。每一个条目都请清楚地显示为`<Supe
- get_absolute_url(): 如果你喜欢在网站中的admin视图和对象详细视图之间切换，该属性会很方便的。如果
- ordering: 如果没有这个元选项，你的条目会在数据数据库返回后以任意顺序出现。你也可以想象一下，如果你
- verbose_name: 如果你忽略该属性，模型的名称会从`CamelCase`转换到`camelcase`。这个例子中，“sup
- verbose_name_plural: 再者，忽略该选项能够给你带来比较有趣的结果。因为Django简单地将一个‘s’预加

这里建议你定义前面的 `Meta` 属性和方法，而不仅仅是只用于admin接口，而且也是为了在shell中和日志文件中，等等中更好的表现内容。

当然，你也可以像下面这样，通过创建一个 `ModelAdmin` 类来进一步改进在admin里的显示：

```
  # admin.py
class SuperHeroAdmin(admin.ModelAdmin):
    list_display = ('name', 'added_on')
    search_fields = ["name"]
    ordering = ["name"]
admin.site.register(models.SuperHero, SuperHeroAdmin)
```

我们来看看这些更为严密的选项：

- list-display: 该选项在一个表格形式的表单中该显示模型实例。它显示每个独立可排序列的字段。如果你希望看到模型的多个属性，该选项是非常理想的。
- search_fields: 该选项在列表上面显示一个搜索框。任何的输入的搜索项都可以搜索到对应的引用字段。因此，仅有CharField或者TextField这样的文本字段被引用。
- ordering: 该选项优先于模型的默认顺序。在admin后台管理中选择一个不同的顺序时，会很有用的。

图片：略

前面的截图插入内容为：

- 插入内容1: 不使用str或者Meta属性
- 插入内容2: 使用增强的模型meta属性
- 插入内容3: 使用定制的ModelAdmin

这里我们仅仅提到了一个常用的amdin选项子集。某些类型的网站会重度地使用admin接口。在这样地情况下，这里强烈建议你彻彻底底搞明白Django文档的admin部分。

不应该让所有人都成为admin

Since admin interfaces are so easy to create, people tend to misuse them. Some give early users admin access by merely turning on their 'staff' flag. Soon such users begin making feature requests, mistaking the admin interface to be the actual application interface.

因为admin接口很轻松就可以创建了，所以有可能被滥用。

Unfortunately, this is not what the admin interface is for. As the flag suggests, it is an internal tool for the staff to enter content. It is production-ready but not really intended for the end users of your website. It is best to use admin for simple data entry. For example, in a project

I had reviewed, every teacher was made an admin for a Django application managing university courses. This was a poor decision since the admin interface confused the teachers.

不幸的是这不是admin接口的本来目的。

The workflow for scheduling a class involves checking the schedules of other teachers and students. Using the admin interface gives them a direct view of the database. There is very little control over how the data gets modified by the admin.

So, keep the set of people with admin access as small as possible. Make changes via admin sparingly, unless it is simple data entry such as adding an article's content.

因此，你要保持能够访问admin人群数量尽可能少。通过admin操作地变更要谨慎，除非是添加一篇文章内容这样地简单数据条目操作。

> 提示
>
> 最佳实践 不要让admin方法终端用户。

Ensure that all your admins understand the data inconsistencies that can arise from making changes through the admin. If possible, record manually or use apps, such as django-audit-loglog that can keep a log of admin changes made for future reference.

确保

In the case of the university example, we created a separate interface for teachers, such as a course builder. These tools will be visible and accessible only if the user has a teacher profile.

Essentially, rectifying most misuses of the admin interface involves creating more powerful tools for certain sets of users. However, don't take the easy (and wrong) path of granting them admin access.

admin接口的定制

开箱即用单admin接口对于准备使用它的人来说非常有用。不幸的是，很多人都假设改变Django的admin肯定非常困难，然后就撒手不管了。实际上，admin是属于极其易于定制的，它的外观可以用最小的努力就得以改变。

改变标题

Many users of the admin interface might be stumped by the heading—Django administration. It might be more helpful to change this to something customized such as MySite admin or something cool such as SuperBook Secret Area.

很多admin用户或许被标题——Django administration给难住了。

要改变标题是很容易的。在站点的urls.py中添加下面这行内容就好了：

```
admin.site.site_header = "SuperBook Secret Area"
```

改变基本样式

几乎所有的admin页面都扩展自叫做admin/base_site.html都公共基本模板。这意味着只需用到少量都HTML和CSS的知识，因此你可以定制所有的排序以改变admin接口的外观和视觉。

先简单地在任意地模板目录中创建一个名称为admin的目录。然后，从Django源目录复制文件base_site.html，并按照自己的需要做相应的变更。如果你不知道模板的位置，那么在Django shell中运行下面的命令就是了：

```
>>> from os.path import join
>>> from django.contrib import admin
>>> print(join(admin.__path__[0], "templates", "admin"))
```

例如，定制admin的基础模板，你可以改变admin接口的整个字体为谷歌字体“Special Elite”，谷歌的这个字体看上去非常的厚重。你需要使用以下内容在项目中的模板目录中添加一个文件admin/base_site.html；

```
{% extends "admin/base.html" %}
{% block extrastyle %}
   <link href='http://fonts.googleapis.com/css?family=Special+Elite'
rel='stylesheet' type='text/css'>
   <style type="text/css">
    body, td, th, input {
      font-family: 'Special Elite', cursive;
} </style>
{% endblock %}
```

该代码通过添加一个附加的样式表来重写与字体相关的样式，而且附加的样式会应用于每个admin的页面。

添加富文本编辑器

有时候，你需要在admin接口中使用JavaScript代码。常见的一个需求就是对TextField使用CKEditor这样的HTML编辑器。

在Django中有多种实现这个编辑器的方法，例如，对ModleAdmin类使用一个Media内部类。不过，我发现扩展admin的change_form模板是最方便的方法。

例如，假如你拥有一个称作Posts的应用，那么你需要去在template/admin/posts/directory目录之内新建一个称作change_form.html的文件。如果你需要在这个应用内的任意模型中显示CKEditor，那么这个文件的内容是这个样子的：

```
/home/arun/env/sbenv/lib/python3.4/site-packages/django/contrib/admin/templates/admin
```

该文件中的最后一行是所有admin模板中的位置所在。你可以重写或者扩展这些模板中的任何一个。可以参考下一小节的扩展模板的例子。

```
{% extends "admin/change_form.html" %}
{% block footer %}
 {{ block.super }}
 <script src="//cdn.ckeditor.com/4.4.4/standard/ckeditor.js"></
script>
<script> CKEDITOR.replace("id_message", {
    toolbar: [
    [ 'Bold', 'Italic', '-', 'NumberedList', 'BulletedList'],],
    width: 600,
  });
 </script>
 <style type="text/css">
  .cke { clear: both; }
 </style>
{% endblock %}
```

高亮的部分是用于我们希望将一个普通的文本输入框加强为富文本编辑器的表单元素的自动创建的ID。这些脚本和样式眼睛被添加到了footer块，这样表单元素可以在自身被改变之前，于DOM中创建。

使用Bootstrap主题的admin

总的来说，admin接口已经设计非常好了。不过，由于它是在2006年设计的，而且是为了显示效果的通用性做出的设计。因此，它没有适应mobile设备，或者是拥有其他的已经成为今日事实标准的细节部分。

毫不奇怪的是admin定制中最常见要求是确定是否可以继承Bootstrap。有多个包可以实现这个需求，比如django-admin-bootstrapped或者djangosuit。

这些包提供了开箱即用的基于Bootstrap主题的模板，而不是你自己去重新编写所有的admin模板。因为基于Bootstrap，所以它们拥有响应式功能，而且包含了多种部件和组件。

彻底检查

admin接口也已经在我们的尝试下完全的重写了。Grappelli是一个非常流行皮肤，它能够利用功能扩展Django admin，比如自动查询和折叠嵌套。使用django-admin-tools，你可以获得一个可定制的面板和工具栏。

There have been attempts made to completely rewrite the admin, such as django-admin2 and nexus, which did not gain any significant adoption. There is even an official proposal called AdminNext to revamp the entire admin app. Considering the size, complexity, and popularity of the existing admin, any such effort is expected to take a significant amount of time.

这里也有完全重写admin的尝试，比如django-admin2和nexus，它们不会着重使用的。

保护admin

The admin interface of your site gives access to almost every piece of data stored. So, don't leave the metaphorical gate lightly guarded. In fact, one of the only telltale signs that someone runs Django is that, when you navigate to http://example. com/admin/, you will be greeted by the blue login screen.

网站的admin接口几乎访问了每一块存储的数据。因此，不要留下缺少保护的后门。实际上，

在生产环境中，我们建议你将这个地址改为不太显眼的地址。在项目的根urls.py中尽可能简单地变更该行：

```
url(r'^secretarea/', include(admin.site.urls)),
```

一个稍微更加成熟的做法是在默认位置使用假的admin站点或者蜜罐（参见第三方包`django-admin-honeypot`）。不过，最好的选择对admin站点范围内使用HTTPS，因为常规的HTTP会把所有的数据以明文格式发送到网络中去。

检查web服务器的文档，看看如何为到admin的请求设置HTTPS。在Nginx上面，设置这个连接方式非常的简单，涉及到的有指定SSL认证位置。最后，将所有到admin页面的HTTP请求重定向到HTTPS，现在可以安生地睡个好觉了。

The following pattern is not strictly limited to the admin interface but it is nonetheless included in this chapter, as it is often controlled in the admin.

模式— 功能标识

遇到的问题：对用户发布的新功能，以及在生产环境中部署的对应代码都应当是互相独立的。

解决方法：在部署之后，使用功能标识有选择性地启动或者禁用功能。

问题细节

Rolling out frequent bug fixes and new features to production is common today. Many of these changes are unnoticed by users. However, new features that have significant impact in terms of usability or performance ought to be rolled out in a phased manner. In other words, deployment should be decoupled from a release.

现如今，惯常的bug修复和新功能在生产环境中是很常见的。对于用户这些改变中很多的改变是不被通知的。不过，

Simplistic release processes activate new features as soon as they are deployed. This can potentially have catastrophic results ranging from user issues (swamping your support resources) to performance issues (causing downtime).

过于简单发布过程

Hence, in large sites it is important to decouple deployment of new features in production and activate them. Even if they are activated, they are sometimes seen only by a select group of users. This select group can be staff or a sample set of customers for trial purposes.

因此，在生产环境中对于大型站点来说最重要的是解构新功能的部署，并激活这些新功能。即使这些新功能被激活了，它们也只是仅仅对被选择了的用户可见。这个被挑选出来的组可以是站点注册成员，也可以是一组简单的出于试验目的而存在的用户。

解决方法细节

Many sites control the activation of new features using Feature Flags. A feature flag is a switch in your code that determines whether a feature should be made available to certain customers.

很多网站的新功能激活是透过功能标识实现的。功能标识是一个代码中的可以决定一个功能是否对某些用户开放的开关。

Django有多个提供功能标识的包，比如 gargoyle 和 django-waffle。这些包在网站的数据库中存储功能标识。他们能够透过admin接口或者管理命令进行激活或者失效。因此，每一种环境（生产、测试、开发、等等）都可以拥有属于自己的一组激活功能。

Feature flags were originally documented, as used in Flickr (See http://code.flickr.net/2009/12/02/flipping-out/). They managed a code repository without any branches, that is, everything was checked into the mainline. They also deployed this code into production several times a day. If they found out that a new feature broke anything in production or increased load on the database, then they simply disabled it by turning that feature flag off.

功能旗帜是得到原生的文档支持，一如用在Flickr。它们不用任何分支管理代码仓库，即，一切内容都记录到主线中。它们在一天可以多次部署到生产环境中。如果在生产环境中发现新的功能破坏了任何其他东西，或者增加了数据库的负载，那么它们都通过关闭功能旗帜来简单的禁用。

Feature flags can be used for various other situations (the following examples use django-waffle):

功能标识可以用于多种情况（下面的例子使用的是django-waffle）：

- Trials: A feature flag can also be conditionally active for certain users. These can be your own staff or certain early adopters than you may be targeting as follows:
- 试用： 功能标识也可以根据条件针对部分用户激活。
  如下，这些用户可以是站点的注册成员，或者某些你想指定监护人：

```
def my_view(request):
    if flag_is_active(request, 'flag_name'):
    # Behavior if flag is active.
```

Sites can run several such trials in parallel, so different sets of users might actually have different user experiences. Metrics and feedback are collected from such controlled tests before wider deployment.

站点可以平行的运行多个试用，这样不同组的用户实际上可以拥有不同的用户体验。在大范围部署之前，可以从这里可控制的测试中收集质量和反馈。

- A/B testing: This is quite similar to trials except that users are selected randomly within a controlled experiment. This is quite common in web design to identify which changes can increase the conversion rates. This is how such a view can be written:
- A/B测试：该测试很类似于体验测试，除了用户在被控制的试验中随机地选择用户。对于web设计来说识别出哪个变更能够增加转换速率是相当常见的。这也展示这样的一个视图是如何编写的：

```
def my_view(request):
    if sample_is_active(request, 'design_name'):
    # Behavior for test sample. 针对测试例子的具体行为
```

- Performance testing: Sometimes, it is hard to measure the impact of a feature on server performance. In such cases, it is best to activate the flag only for a small percentage of users first. The percentage of activations can be gradually increased if the performance is within the expected limits.
- 性能测试：有时候，很难去测量服务器上一个功能性能影响。这类例子中，最好是首先仅对一小部分激活旗帜。如果性能存在未预料地的限制，激活百分比可以逐渐地增加。

- Limit externalities: We can also use feature flags as a site-wide feature switch that reflects the availability of its services. For example, downtime in external services such as Amazon S3 can result in users facing error messages while they perform actions, such as uploading photos.
- 扩展性的限制：我们也可以使用功能旗帜

When the external service is down for extended periods, a feature flag can be deactivated that would disable the upload button and/or show a more helpful message about the downtime. This simple feature saves the user's time and provides a better user experience:

当扩展服务因为扩展周期而关闭时，新的功能旗帜被取消激活将会禁用上传按钮同时／或者显示关于关闭时间更为有帮助的消息。这个简单的功能保存了用户的时间并提供了更好的用户体验：

```
def my_view(request):
    if switch_is_active('s3_down'):
    # Disable uploads and show it is downtime 禁用上传并在被禁用时显示
```

The main disadvantage of this approach is that the code gets littered with conditional checks. However, this can be controlled by periodic code cleanups that remove checks for fully accepted features and prune out permanently deactivated features.

这个方法的主要缺点是按照某些条件检查代码会变得垃圾。不过，

总结

本章我们探究了Django的内建应用admin。我们发现了它不仅仅是非常好用的开箱即用，而且可以实现各种定制，以改进它的外观和功能。

下一章，我们将会通过思考多种模式和常见用法来学习在Django中如何更有效的使用表单。

第七章-表单

这一章我们会讨论一下话题：

```
• 表单的工作流程
• 不可靠的输入
• 表单处理类视图
• 表单与CRUD视图
```

我们把Django表单放到一边，来讨论下常规情况下的表单是个什么样子。表单不仅长，而且有着多个需要填充的无趣的页面。可以说表单无所不在。我们每天都用到它。表单支撑了谷歌搜索框到脸书的点赞按钮这所有的一切。

Django把使用表单时产生的验证和描述这类的大量繁重工作给抽象了。它也实现了多种的安全问题的最佳实践。可是，表单在处理自身多个状态之一时也是令人困惑的起因。

表单的工作原理

表单理解起来比较困难，因为它同不止一个请求-响应循环交互。最简单的场景是，你需要展示一个空表单，然后用户来正确地填充和提交表单。另外一种情况是它们输入一些无效的数据，表单需要重复的提交知道整个表单有效为止。

因此，表单表现出多种状态：

- 空表单：在Django中此表单称为未绑定表单
- 已填充表单：Django中该表单称为已绑定表单
- 有错误的已提交表单：该表单称做已绑定表单，但不是有效表单
- 没有错误的已提交表单：该表单称做已绑定且有效的表单

注意用户永远不会见到表单的最后状态。他们不必如此。提交的有效表单应当把用户带到表单提交成功之后的页面。

Django中的表单

通过总结它们到一个层次，Django的`form`类每个字段的状态，以及表单自身。表单拥有两个重要的状态属性，一如下面所示：

- *is_bound*: 如果返回值为假，则它是一个未绑定的表单，即，新的空表单，或者默认的字段值。如果返回值为真，表单被绑定，即，至少有一个用户设置的字段。

- *is_valid()*: 如果返回值为真，已绑定表单中的所有字段都拥有有效的数据。如果返回假，至少有一个字段中存在一部分无效数据，或者表单未被绑定。

举例来说，想象一下你需要一个接受用户名字和年龄的简单表单。这个类可以这样来定义：

```
# forms.py
from django import forms


class PersonDetailsForm(forms.Form):
    name = forms.CharField(max_length=100)
    age = forms.IntegerField()
```

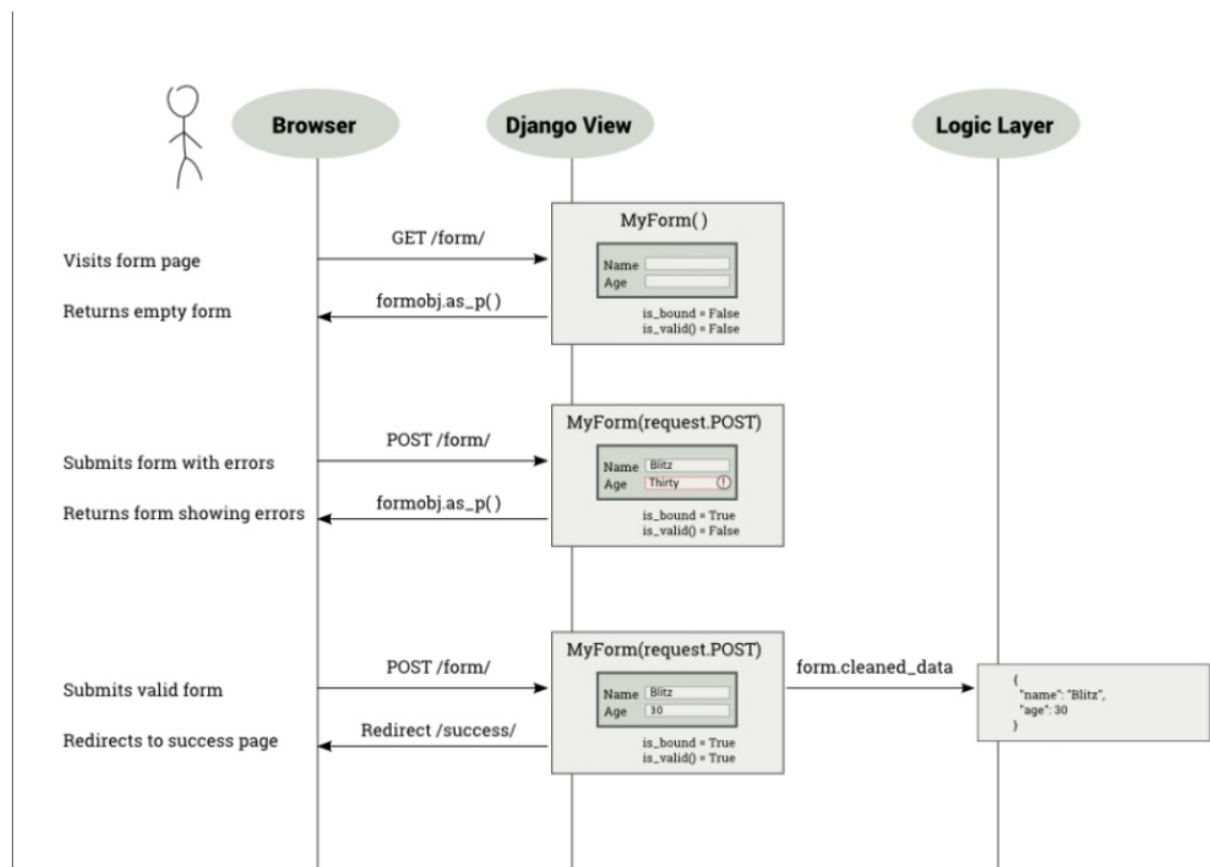
该类可以以绑定或者不绑定方式来初始化，一如下面代码所示：

```
>>> f = PersonDetailsForm()
>>> print(f.as_p())
<p><label for="id_name">Name:</label> <input id="id_name" maxlength="100"
name="name" type="text" /></p>
<p><label for="id_age">Age:</label> <input id="id_age" name="age"
type="number" /></p>
>>> f.is_bound
False
>>> g = PersonDetailsForm({"name": "Blitz", "age": "30"})
>>> print(g.as_p())
<p><label for="id_name">Name:</label> <input id="id_name" maxlength="100"
name="name" type="text" value="Blitz" /></p>
<p><label for="id_age">Age:</label> <input id="id_age" name="age"
type="number" value="30" /></p>
>>> g.is_bound
 True
```

要注意HTML是如何表现改变以包括它们中已绑定数据的值属性。

表单可以只在你创建表单对象时才被绑定，即，在构造器中。用户如何在类字典对象的最后面输入每个表单字段的值？

要想解决这个问题，你需要理解用户是如何与表单交互的。在下面的的图表中，用户打开用户账户表单，首先是正确地填充，并提交它，然后用有效信息重新提交表单：



如前面的表单所示，当用户提交表单时，在 `request.POST` （它是一个 `QueryDict` 的实例）内部所有可调用的视图获取到全部的表单数据。表单使用类字典对象——以这种方式引用是因为它的行为类似于字典，来初始，并拥有一点额外的功能。

表单可以通过两种不同的方式来定义以发送数据表单： `GET` 或者 `POST` 。表单使用 `METHOD="GET"` 定义以发送以URL编码的表单数据，例如，当你提交谷歌搜索时，URL取得表单输入，即，搜索字符串显式地嵌入，比如 `?q=Cat+Pictures` 就是如此。 `GET` 方法用来幂等表单，它不会对世界状态做出任何的最新改变。（不要太过于迂腐，多次处理有同样的效果就像一次处理）。大多数情况下，这意味着它只在重新取回数据时被用到。

不过，不计其数的表单都是使用 `METHOD="POST"` 来定义的。这样，表单数据会一直发送HTTP请求的主体部分，而且它们对于用户来说是不可见的。它们被用于任何涉及到边际效应的事情，比如存储或者更新数据。

视你所定义表单类型的不同，当用户提交表单时，视图会重新取回 `request.GET` 或者 `request.POST` 中的表单数据。如同早前咱么提到的那样，它们中的哪一个都类似于字典。因此，你可以传递它到表单类构造器以获取绑定的 `form` 对象。

> 注释
>
> **The Breach** Steve was curled up and snoring heavily in his large three-seater couch. For the last few weeks, he had been spending more than 12 hours at the office, and tonight was no exception. His phone lying on the carpet beeped. t first, he said

> soething incoherently, still deep in sleep. Then, it beeped again and again, in increasing urgency.
>
> By the fifth beep, teve awoke with a start. He frantically searched all over his couch, and finally located his phone. The screen showed a brightly colored bar chart. Every bar seemed to touch the high line except one. He pulled out his laptop and logged into the SuperBook server. The site was up and none of the logs indicated any unusual activity. However, the external services didn't look that good.
>
> The phone at the other end seemed to ring for eternity until a croaky voice answered, Hello, teve? Half an hour later, acob was able to ero down the proble to an unresponsive superhero verification service. Isn't that running on auron? asked teve. There was a brief hesitation. "I am afraid so," replied Jacob.
>
> Steve had a sinking feeling at the pit of his stomach. Sauron, a ainfrae application, was their first line of defense against cyber-attacks and other kinds of possible attack. It was three in the morning when he alerted the mission control team. Jacob kept chatting with him the whole time. He was running every available diagnostic tool. There was no sign of any security breach.
>
> Steve tried to calm him down. He reassured him that perhaps it was a temporary overload and he should get some rest. However, he knew that Jacob wouldn't stop until he found what's wrong. He also knew that it was not typical of Sauron to have a temporary overload. Feeling extremely exhausted, he slipped back to sleep.
>
> Next morning, as steve hurried to his office building holding a bagel, he heard a deafening roar. He turned and looked up to see a massive spaceship looming towards him. Instinctively, he ducked behind a hedge. On the other side, he could hear several heavy metallic objects clanging onto the ground. Just then his cell phone rang. It was Jacob. Something had moved closer to him. As Steve looked up, he saw a nearly 10-foot-tall robot, colored orange and black, pointing what looked like a weapon directly down at him.
>
> His phone was still ringing. He darted out into the open barely missing the sputtering shower of bullets around him. He took the call. "Hey Steve, guess what, I found out what actually happened." "I am dying to know," Steve quipped.
>
> "Remember, we had used UserHoller's form widget to collect customer feedback? pparently, their data was not that clean. I ean several serious exploits. Hey, there is a lot of background noise. Is that the T? Steve dived towards a large sign that said "Safe Assembly Point". "Just ignore that. Tell me what happened," he screamed.

> "Okay. So, when our admin opened their feedback page, his laptop must have gotten infected. The worm could reach other systems he has access to, specifically, auron. I ust say acob, this is a very targeted attack. Someone who knows our security system quite well has designed this. I have a feeling something scary is coming our way."
>
> Across the lawn, a robot picked up an SUV and hurled it towards Steve. He raised his hands and shut his eyes. The spinning mass of metal froze a few feet above hi. Iportant call? asked Hexa as she dropped the car. Yeah, please get e out of here, teve begged.

为什么数据需要清理

终究你还是需要从表单获取“干净的数据”。这是否意味着用户输入的值是否是不干净的呢？是的，这里有两个理由。

首先来自外部世界的任何东西都不应该一开始就被信任。恶意用户可以对表单输入所有类别的探测利用，以此破坏网站的安全。因此，任何的表单数据在使用前都必须被净化。

> 提示
>
> *Best Practice* 任何时候都不能信任用户的输入。

第二， `request.POST` 或者 `request.GET` 中的字段值只是字符串而已。即使表单字段定义为整数（比如说，年龄）或者日期（比如说，生日），浏览器也是把这些字段以字符串的形式发送到视图。不可避免的是在加以使用之前你要把它们转换到适当的Python数据类型。 `form` 类在进行清理时会自动地达成约定。

我们来看看实际的例子：

```
>>> fill = {"name": "Blitz", "age": "30"}
>>> g = PersonDetailsForm(fill)
>>> g.is_valid()
    True
>>> g.cleaned_data
    {'age': 30, 'name': 'Blitz'}
>>> type(g.cleaned_data["age"])
    int
```

年龄值作为字符串来传递（或许来自 `request.POST` ）到表单类。验证之后，干净数据包含整数形式的年龄。这完全符合用户的期望。表单试图抽象出字符串的传递，以及对用户给出可以使用的干净的Python数据类型的事实。

显示表单

Django表单也能够帮助你生成表现表单的HTML。它们支持臆中不同的表现形式：`as_p`（作为段落标签），`as_ul`（显示为无序列表项），以及`as_table`（意料之中，显示为表格）。

模板代码生成HTML代码，浏览器渲染这些表现已经总结为下表：

图片：略

注意HTML表现仅给出了表单字段。这样做可以在一个单独的HTML表单中轻松地包含多个Django表单。可是，这样做也意味着模板设计者必须有点儿公式化来写每个表单，一如下面代码所示：

```
<form method="post">
     {% csrf_token %}
     <table>{{ form.as_table }}</table>
     <input type="submit" value="Submit" />
</form>
```

注意，为了使HTML完整的表现，你需要添加`form`标签周添加CSRF令牌环，`table`或者`ul`标签，以及`submit`按钮。

是使用crispy的时候了

在模板中写这么多公式化的表单是很无聊的事情。`django-crispy-forms`包可以写非长干脆利落的来写表单模板。它把所有表现和布局都放到了Django表单自身。因此，你可以写更多的Python代码，更少的HTML。

下面的表展示了csrispy表单的模板标签生成更加完整的表单，而且外观更加的接近原生的Bootstrap风格：

Template	Code
{% crispy_form}	<from method="post"><input type='hidden' name='csrfmiddlewaretoken' valu 了简洁而删去余下的HTML)

那么，你是如何实现更加干脆利落的表单的？你需要安装`django-crispy-forms`包并将它加入到`INSTALLED_APPS`。如果你使用Bootstrap 3，那么你需要在设置中引用：

```
CRISPY_TEMPLATE_PACK = "bootstrap3"
```

表单初始化需要引用类型`FormHelper`的辅助属性。下面的代码以最小化设计，并使用默认的布局：

```
from crispy_forms.helper import FormHelper
from crispy_forms.layout import Submit


   class PersonDetailsForm(forms.Form):
       name = forms.CharField(max_length=100)
       age = forms.IntegerField()

       def __init__(self, *args, **kwargs):
           super().__init__(*args, **kwargs)
           self.helper = FormHelper(self)
           self.helper.layout.append(Submit('submit', 'Submit'))
```

理解CSRF

你也一定注意到了在表单模板中叫做 `CSRF` 的东西。它用来干什么的？它是一种应对针对表单的 `跨站请求伪造（CSRF）` 的保护机制。

它通过注入服务端生成的随机的叫做CSRF令牌的字符工作，这个字符串对用户的session来说是唯一的。每次表单提交时，都有一个包含此令牌的隐藏字段。这个令牌确保表单是由用户生成而且是来自原始站点的，而不是由攻击者使用类似字段创建的假冒表单。

在表单使用 `GET` 方法时并不推荐CSRF令牌，因为 `GET` 行为不应该改变服务器状态。此外，表单通过 `GET` 提交表单会在URL中暴露CSRF令牌。因此URL在登录或者并行嗅探时具有更高的风险，当使用 `POST` 方式最好在表单中使用CSRF。

表单处理类视图

基本上，我们可以通过子类化类视图来处理表单：

```
    class ClassBasedFormView(generic.View):
        template_name = 'form.html'
        def get(self, request):
            form = PersonDetailsForm()
            return render(request, self.template_name, {'form': form})
        def post(self, request):
            form = PersonDetailsForm(request.POST)
            if form.is_valid():
                # 成功的话，我就可以使用了form.cleaned_data。
                return redirect('success')
            else:
                # 无效，重新显示含有错误高亮的表单
                return render(request, self.template_name,
                              {'form': form})
```

上面的代码与我们之前的见到的序列图表相比较。三种应用场景已经被分别处理。

每一个表单都如期望的那样遵守 `Post/Redirect/Get(PRG)` 模式。如果提交的表单发现是无效的，它必须发起一个重定向。这能够阻止重复表单的提交。

不过，这样做是并不十分符合DRY原则的编写。表单类名称和模板名称属性都已经重复了。使用 `FormView` 这样的通用类视图能够表单处理中的冗余部分。下面的代码会给你带来和前面一样的功能而且还少了几行代码：

```
from django.core.urlresolvers import reverse_lazy

class GenericFormView(generic.FormView):
    template_name = 'form.html'
    form_class = PersonDetailsForm
    success_url = reverse_lazy("success")
```

这个例子中我们需要使用 `reverse_lazy`，因为URL模式在视图文件导入时并没有被载入。

表单模式

我们来看一看使用表单时会见到的一些常用模式。

模式-动态表单的生成

问题：自动地添加表单字段或者改变已经声明的表单字段。

解决方案：在表单初始化的时候添加或者改变字段。

问题细节

表单通常以一种将拥有的字段列为类字段的声明方式。不过，有时候我们不能提前知道数量，或者这些字段的类型。这就要求表单能够动态地生成。该模式有时称为 `动态表单` 或者 `运行时的表单生成`。

想象有一个旅客航班登录系统，它允许经济舱机票改签为头等舱。假如头等舱座席有剩余，那么在用户想要乘坐头等舱是就需要有一个额外的选项。不过，这个额外的字段不能够公开，因为它对于全部用户来说是不可见的。这样的动态表单就可以通过该模式来处理。

解决方案细节

每一个表单的实例都有一个叫做字段的 `fields`，它是一个拥有全部字段的字典。可以在运行时对它作出修改。添加或者改变字段可以在表单初始化时完成。

例如，如果我们添加一个到用户详情表单的复选框，只要在表单初始化时命名为“upgrade”的关键字参数为真，那么我们就可以以如下代码来实现它：

```
class PersonDetailsForm(forms.Form):
    name = forms.CharField(max_length=100)
    age = forms.IntegerField()
    def __init__(self, *args, **kwargs):
       upgrade = kwargs.pop("upgrade", False)
       super().__init__(*args, **kwargs)
       # Show first class option? 显示头等舱选项？
       if upgrade:
            self.fields["first_class"] = forms.BooleanField(label="Fly First Class?")
```

现在，我们只需要传递关键字参数 `PersonDetailsForm(upgrade=True)` 以产生一个额外的布尔输入字段（复选框）。

> 注释
>
> 注意，最新引入的关键字参数在我们调用 `super` 以避免 `unexpected keyword` 错误已经被移除，或者去掉。

如果我们对这个例子使用 `FormView` 类，那么我们需要通过重写视图类的 `get_form_kwargs` 方法来传递关键字参数，一如下面代码所示：

```
class PersonDetailsEdit(generic.FormView):
    ...
    def get_form_kwargs(self):
        kwargs = super().get_form_kwargs()
        kwargs["upgrade"] = True
        return kwargs
```

这个模式可以用来在运行时改变任意一个字段的属性，比如它的部件或者辅助文本。它也能够用于模型表单。

在很多情况下，外观所需的动态表单可以通过使用Django表单集合来解决。在页面中当表单需要重复时就要用到它们了。一个典型的表单集合用法是在设计数据的网格视图时一行接一行的添加元素。因此，你不需要使用任意数量的行来创建一个动态表单。你只需依照行来创建表单，使用 `formset_factory` 函数来创建多个行。

模式-用户表单

问题：表单需要根据已经登录的用户来进行定制。

解决方案：传递已登录用户作为关键字参数到表单的构造器。

问题细节

表单可以按用户以不同的形式来表现。某些用户或许不需要填充全部字段，而其他人就需要添加额外的信息。某些情况下，你或许需要对用户资格做些检查，比如，验证他们是否为一个组的成员，以便搞清楚如何构建表单。

解决方案细节

你一定注意到了，你可以使用动态表单生成模式中给出的解决方案来解决这个问题。你只需要将 `request.user` 作为一个关键字参数传递到表单。然而，为了简洁和更富于重复使用的解决方案，我们也可以使用 `django-braces` 包中的mixin。

如前面的例子所示，我们需要对用户展示一个额外的复选框。可是，这仅在用户是VIP组成员时才会被显示。让我们来看看 `PersonDetailsForm` 是如何使用 `django-braces` 的表单mixin `UserKwargModelFormMixin` 来简化的：

```
from braces.forms import UserKwargModelFormMixin

class PersonDetailsForm(UserKwargModelFormMixin, forms.Form):
    ...
    def __init__(self, *args, **kwargs):
       super().__init__(*args, **kwargs)
       # Are you a member of the VIP group?
       # 你是VIP组的成员吗？
       if self.user.groups.filter(name="VIP").exists():
           self.fields["first_class"] = forms.BooleanField(label="Fly First Class?")
```

注意 `self.user` 如何通过mixin去掉用户关键字参数被自动地设为可用。

对应到表单mixin，有一个成为 `UserFormKwargsMixin` 的视图mixin，它需要添加到视图，使用 `LoginRequiredMixin` 以确保只有已经登录的用户可以访问该视图：

```
class VIPCheckFormView(LoginRequiredMixin, UserFormKwargsMixin,
generic.FormView):
   form_class = PersonDetailsForm
    ...
```

现在用户参数自动地传递到表单 `PersonDetailsForm` 。

在 `django-braces` 中检查其他的表单mixin，比如拿 `FormValidMessageMixin` 来说，它就是常见的表单使用模式的一个现成的方案。

模式-一个视图的多个表单行为

问题：在一个独立的视图或者页面中处理多个表单行为。

解决方案：表单可以使用独立的视图来处理表单提交，或者识别基于提交按钮名称的表单。

问题细节

Django使合并多个拥有相同行为的表单相当地简单，例如，单独的一个提交按钮。不过，大多数的页面需要在同样的页面上显示多个行为。例如，你或许想要用户在同一页面的两个不同表单中订阅或取消订阅一个新闻简报。

不过，Django的 `FormView` 被设计成只处理一个表单对应一个视图的场景。很多的其他通用类视图也共享该假定。

解决方案细节

处理多个表单有两个办法：分离视图和独立视图。首先我们来看下第一个方法。

独立的行为分离视图

依据视图的行为为每个表单指定不同的视图，是一个相当简单的办法。例如，订阅和取消订阅表单。刚好有两个独立的视图类来处理他们各自表单的 `POST` 方法。

独立的相同视图

可能你发现了分割视图来处理表单是没有必要的，抑或发现了使用一个公共视图来处理逻辑上相关连的表单为更加简洁明了。

当对多个表单使用相同的视图类时，其挑战在于标识哪个表单来处理POST行为。这里，我们利用了事实上的优势，提交按钮的名称和值同时被提交了。假如提交按钮在表单中命名唯一，那么处理时表单就可以被标识出来。

这里，我们定义一个使用crispy表单的订阅器，这样我们就可以的命名提交按钮了：

```
class SubscribeForm(forms.Form):
    email = forms.EmailField()

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.helper = FormHelper(self)
        self.helper.layout.append(Submit('subscribe_butn', 'Subscribe'))
```

取消订阅表单类 `UnSubscribeForm` 以完全相同的方式来定义（因此，这里我们就省略不写出来了），除了 `Submit` 按钮被命名为 `unscribe_butn` 之外。

因为 `FormView` 被设计成单视图，我们会使用一个简单的类视图，即，`TemplageView` ，来做为视图的基类。我们来看一看视图定义以及 `get` 方法：

```
from .forms import SubscribeForm, UnSubscribeForm

class NewsletterView(generic.TemplateView):
    subcribe_form_class = SubscribeForm
    unsubcribe_form_class = UnSubscribeForm
    template_name = "newsletter.html"

    def get(self, request, *args, **kwargs):
        kwargs.setdefault("subscribe_form", self.subcribe_form_class())
        kwargs.setdefault("unsubscribe_form", self.unsubcribe_form_class())
        return super().get(request, *args, **kwargs)
```

`TemplateView` 类的关键字参数可以方便地插入进模板上下文。我们仅在表单实例不存在时，利用 `setdefault` 字典方法的帮助来创建其中一个表单的实例。我们很快就可以看到为什么要这样做。

接下来，我们来看看POST方法，它处理了表单的提交：

```
    def post(self, request, *args, **kwargs):
        form_args = {
            'data': self.request.POST,
            'files': self.request.FILES,
        }

        if "subscribe_butn" in request.POST:
            form = self.subcribe_form_class(**form_args)
            if not form.is_valid():
                return self.get(request,
                                subscribe_form=form)
            return redirect("success_form1")
        elif "unsubscribe_butn" in request.POST:
            form = self.unsubcribe_form_class(**form_args)
            if not form.is_valid():
                return self.get(request,
                                unsubscribe_form=form)
            return redirect("success_form2")
        return super().get(request)
```

首先，表单的关键字参数，比如数据和文件，就是在 `form_args` 字典中产生的。接下来，第一个表单的 `Submit` 按钮的存在是用来检查 `request.POST` 。假如发现了按钮的名称，那么第一个表单就被初始化。

如果表单验证失败，那么响应通过第一个表单实例的 `GET` 方法被创建的方法就会返回。同样地，我们查找第二个表单的提交按钮以检查第二个表单是否被提交。

相同视图中的相同表单的实例可以通过表单前缀以相同方式来实现。你可以使用 `SubscribeForm(prefix="offers")` 这样的前缀参数来实例化一个表单。比如实例利用给出的参数作为前缀加入到所有的表单字段，实际上当作一个表单的命名空间来使用。

模式-CRUD视图

问题：公式化的对模型编写CRUD接口是在重复相同的事。

解决方案：使用类的通用视图来编辑视图。

问题细节

在大多数的web应用中，大约百分之80的时间被用来写，创建，读取，更新以及删除（CRUD）数据库的接口。例如，基本上，Twitter就涉及到了创建和读取其他用户的推文。这里，推文是可以被维护和存储的数据库对象。

要是从零开始写这样的接口实在是乏味至极。如果CRUD接口可以自动地从模型类创建，那么这个模式可以轻松地管理。

解决方案细节

Django利用了一个四个通用类视图的组简化了创建CRUD视图的过程。如下，它们可以被映射到其自身像对应的操作：

- CreateView: 该视图显示一个空白表单以创建一个新的对象。
- DetailView: 该视图通过读取数据库来展示一个对象的细节。
- UpdateView: 该视图被允许通过一个预先生成的表单来更新一个对象的细节。
- DeleteView: 该视图像是一个确认页面，并准许删除对象。

让我们来看一个简单的例子。我们拥有一个包含重要日期的模型，它关系到使用网站的每一个用户的利益。我们需要构建简单的CRUD接口，这样任何人都可以查看，修改这些日期。我们来看下 `Importantdate` 模型的内容：

```
# models.py
   class ImportantDate(models.Model):
       date = models.DateField()
       desc = models.CharField(max_length=100)
       def get_absolute_url(self):
           return reverse('impdate_detail', args=[str(self.pk)])
```

`get_absolute_url()` 方法被 `CreateView` 和 `UpdateView` 类在对象成功创建或者更新之后使用。它已经被路由到了对象的 `DetailView` 。

这些CRUD视图足够的简单，因此它们不解自明的，一如以下代码所示：

```
 # views.py
from django.core.urlresolvers import reverse_lazy
from . import forms


class ImpDateDetail(generic.DetailView):
    model = models.ImportantDate

class ImpDateCreate(generic.CreateView):
    model = models.ImportantDate
    form_class = forms.ImportantDateForm

class ImpDateUpdate(generic.UpdateView):
    model = models.ImportantDate
    form_class = forms.ImportantDateForm

class ImpDateDelete(generic.DeleteView):
    model = models.ImportantDate
    success_url = reverse_lazy("impdate_list")
```

这些通用视图中，模型类只是强制成员被引用。不过，在 `DeleteView` 的情形下， `success_url` 函数需要很好地的应用。这是因为 `get_absolute_url` 删除之后再也不能够用来找到在什么地方重定向用户。

定义 `form_class` 属性并非是强制的。如果它被省略，与 `ModelForm` 方法相当的模型会被创建。不过，我们想要创建自己的模型表单以利用crispy表单，一如下面代码所示：

```
# forms.py
from django import forms
from . import models
from crispy_forms.helper import FormHelper
from crispy_forms.layout import Submit


class ImportantDateForm(forms.ModelForm):
    class Meta:
        model = models.ImportantDate
        fields = ["date", "desc"]
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.helper = FormHelper(self)
        self.helper.layout.append(Submit('save', 'Save'))
```

要感谢crispy表单，我们在模板中需要非常少的HTML装饰以构建这些CRUD表单。

> ###### 注释 注意明确地引用 `ModelForm` 方法的字段就是最佳实践，在未来的发行版中这也将成为强制规定。

默认，模板的路径基于视图类和模型的名称。为了简洁起见，此处我们省略了模板的源码。注意我们可以对 `CreateView` 和 `UpdateView` 使用相同的表单。

最后，我们来看看 `urls.py` ，这里所有东西都连接起来了：

```
url(r'^impdates/create/$',
        pviews.ImpDateCreate.as_view(), name="impdate_create"),
    url(r'^impdates/(?P<pk>\d+)/$',
        pviews.ImpDateDetail.as_view(), name="impdate_detail"),
    url(r'^impdates/(?P<pk>\d+)/update/$',
        pviews.ImpDateUpdate.as_view(), name="impdate_update"),
    url(r'^impdates/(?P<pk>\d+)/delete/$',
        pviews.ImpDateDelete.as_view(), name="impdate_delete"),
```

Django的通用是我们开始为模型创建CRUD视图的一种了不起的方式。仅需少少几行代码，你就可以获得经过良好测试的模型表单和视图，而不用自己来做重复乏味的工作。

总结

在这一章，我们见过了web表单如何被创建，以及在Django中如何利用表单类将它们抽象。我们也即拿过了在使用表单时，利用多种技术和模式去节省时间。

在下一章，我们来看一看在使用旧版本的Django代码库时用到的系统化的方法，以及当我们碰到用户需要时如何的加强它。

第八章-处理旧版本代码

在本章我们将讨论一下话题：

- 阅读Django代码
- 探索相关文档
- 增量变更还是完全重写
- 改变代码之前编写测试
- 旧版本数据库的集成

It sounds exciting when you are asked to join a project. Powerful new tools and cutting-edge technologies might await you. However, quite often, you are asked to work with an existing, possibly ancient, codebase.

加入项目时你可能很感兴趣。

To be fair, Django has not been around for that long. However, projects written for older versions of Django are sufficiently different to cause concern. oeties, having the entire source code and documentation might not be enough.
公平起见，Django并没有一直这样做。不过，项目

Yeah, said Brad, Where is Hart? Mada hesitated and replied, "Well, he resigned.
Being the head of IT security, he took moral responsibility of the perimeter breach." Steve, evidently shocked, was shaking his head. "I am sorry," she continued, "But I have been assigned to head SuperBook and ensure that we have no roadblocks to meet the new deadline."

There was a collective groan. Undeterred, Madam O took one of the sheets and began, "It says here that the Remote Archive module is the most high-priority item in the incomplete status. I believe Evan is working on this." If you are asked to recreate the environment, then you might need to fumble with the configuration, database settings, and running services locally or on the network. There are so many pieces to this puzzle that you might wonder how and where to start.

Understanding the Django version used in the code is a key piece of information. As Django evolved, everything from the default project structure to the recommended best practices have changed. Therefore, identifying which version of Django was used is a vital piece in understanding it.

理解Django的版本再代码中的使用是很关键的信息。随着Django的进化，所有来自默认项目结构的建议最佳实践都发生了变化。因此，认出Django在使用的是哪一个版本是理解这个框架中重要的一环。

> **Change of Guards**
>
> Sitting patiently on the ridiculously short beanbags in the training room, the SuperBook team waited for Hart. He had convened an emergency go-live meeting. Nobody understood the "emergency" part since go live was at least 3 months away.
>
> Madam O rushed in holding a large designer coffee mug in one hand and a bunch of printouts of what looked like project timelines in the other. Without looking up she said, "We are late so I will get straight to the point. In the light of last week's attacks, the board has decided to summarily expedite the SuperBook project and has set the deadline to end of next onth. ny questions?
>
> Yeah, said Brad, Where is Hart? Mada hesitated and replied, "Well, he resigned. Being the head of IT security, he took moral responsibility of the perimeter breach." Steve, evidently shocked, was shaking his head. "I am sorry," she continued, "But I have been assigned to head SuperBook and ensure that we have no roadblocks to meet the new deadline."
>
> There was a collective groan. Undeterred, Madam O took one of the sheets and began, "It says here that the Remote Archive module is the most high-priority item in the incomplete status. I believe Evan is working on this."
>
> "That's correct," said Evan from the far end of the room. "Nearly there," he smiled at others, as they shifted focus to him. Madam O peered above the rim of her glasses and smiled almost too politely. "Considering that we already have an extremely well-tested and working Archiver in our Sentinel code base, I would recommend that you leverage that instead of creating another redundant system."
>
> "But," Steve interrupted, "it is hardly redundant. We can improve over a legacy archiver, can't we? If it isn't broken, then don't fix it, replied Madam O tersely. He said, "He is working on it," said Brad almost shouting, What about all that work he has already finished?
>
> van, how uch of the work have you copleted so far? asked , rather impatiently. "About 12 percent," he replied looking defensive. veryone looked at hi incredulously. What? That was the hardest percent" he added.
>
> O continued the rest of the meeting in the same pattern. Everybody's work was reprioritied and shoehorned to fit the new deadline. s she picked up her papers, readying to leave she paused and removed her glasses.

> "I know what all of you are thinking... literally. But you need to know that we had no choice about the deadline. All I can tell you now is that the world is counting on you to meet that date, somehow or other." Putting her glasses back on, she left the room.
>
> "I am definitely going to bring my tinfoil hat," said Evan loudly to himself.

查找Django的版本信息

理论上，每个项目在根目录都拥有一个requirements.txt文件，或者 一个setup.py文件，这个文件将表明Django再项目中要使用的是哪一个版本。让我们来看看于此相关的文件片段：

```
Django==1.5.9
```

注意，版本数字已经完全提过了的称作链（相对于Django>=1.5.9）。对每一个包都进行链接是一个好习惯，因为它能够减少人们的疑问，让你更为确定版本信息。

Unfortunately, there are real-world codebases where the requirements.txt file was not updated or even completely missing. In such cases, you will need to probe for various telltale signs to find out the exact version.

不幸的是，

激活虚拟环境

In most cases, a Django project would be deployed within a virtual environment. Once you locate the virtual environment for the project, you can activate it by jumping to that directory and running the activated script for your OS. For Linux, the command is as follows:

再很多情况下，Django的项目会部署在一个虚拟环境中。一旦你找出了项目的虚拟环境，你可以通过跳过这个目录，并为系统运行激活的脚本。对于Linux来说，可以使用如下命令：

```
$ source venv_path/bin/activate
```

Once the virtual environment is active, start a Python shell and query the Django version as follows:

只要虚拟环境一激活，你就可以启动Python终端，然后像我这样查询Django的版本：

```
$ python
>>> import django
>>> print(django.get_version())
1.5.9
```

The Django version used in this case is Version 1.5.9.

本例中使用的Django版本是 1.5.9.

Alternatively, you can run the manage.py script in the project to get a similar output:

可选择的是，你可以在项目中运行脚本 manage.py 以获取类似的输出内容：

```
$ python manage.py --version
1.5.9
```

However, this option would not be available if the legacy project source snapshot was sent to you in an undeployed form. If the virtual environment (and packages) was also included, then you can easily locate the version number (in the form of a tuple) in the `__init__.py` file of the Django directory. For example:

不过呢，要是在未部署的情况下，之前遗留的项目源码镜像被发送给你了，那么这个选项是不可用的。如果虚拟环境（以及包）被包括在内，那么你在Django目录中的 `__init__.py` 文件简单地找到版本号（以元组地形式出现）。例如：

```
$ cd envs/foo_env/lib/python2.7/site-packages/django
$ cat __init__.py
VERSION = (1, 5, 9, 'final', 0)
...
```

If all these methods fail, then you will need to go through the release notes of the past Django versions to deterine the identifiable changes form exaple, the AUTH_PROFILE_MODULE setting was deprecated since Version 1.5) and match them to your legacy code. Once you pinpoint the correct Django version, then you can move on to analyzing the code.

如果，所有这些方法都不管用，那么你需要

文件都放在哪里了？这可不是PHP啊

最大的一个困难点是用到的场合，特别是如果你来自PHP或者APS.NET的世界，即，源文件并不位于web服务器的文档根目录，而目录却通常命名为 `wwwroot` 或者 `public_html` 。此外，代码的目录结构和网站的URL结构之间并没有直接的关系。

In fact, you will find that your Django website's source code is stored in an obscure path such as /opt/webapps/my-django-app. Why is this? ong any good reasons, it is often ore secure to ove your confidential data outside your public webroot. This way, a web crawler would not be able to accidentally stumble into your source code directory.

实际上，你会发现自己的Django站点源码被存储在了一个使人难以理解的的路径中，比如，/opt/webapps/my-django-app。为什么会这样？

As you would read in the Chapter 11, Production-ready the location of the source code can be found by exaining your web server's configuration file. Here, you will find either the environment variable DJANGO_SETTINGS_MODULE being set to the module's path, or it will pass on the request to a WI server that will be configured to point to your project.wsgi file.
和你之前在第十一章读到的那样，

Starting with urls.py

Even if you have access to the entire source code of a Django site, figuring out how it works across various apps can be daunting. It is often best to start from the root urls.py URLconf file since it is literally a ap that ties every request to the respective views.

即使你访问了整个Django站点的源代码，要搞清楚url如何与多个应用交互是令人畏惧的。

With normal Python programs, I often start reading from the start of its execution—say, from the top-level main module or wherever the **main** check idiom starts. In the case of Django applications, I usually start with urls.py since it is easier to follow the ow of execution based on various URL patterns a site has.

In Linux, you can use the following find command to locate the settings.py file and the corresponding line specifying the root urls.py:

```
$ find . -iname settings.py -exec grep -H 'ROOT_URLCONF' {} \;
./projectname/settings.py:ROOT_URLCONF = 'projectname.urls'
$ ls projectname/urls.py
projectname/urls.py
```

Jumping around the code Reading code sometimes feels like browsing the web without the hyperlinks. When you encounter a function or variable defined elsewhere, then you will need to jup to the file that contains that definition. oe IDs can do this autoatically for you as long as you tell it which files to track as part of the project.

If you use Emacs or Vim instead, then you can create a TAGS file to quickly navigate between files. Go to the project root and run a tool called *Exuberant Ctags* as follows:

如果你使用的是Emacs或者Vim，那么你可以创建一个TAG文件来快速地在多个文件之间进行浏览。如下，切换目录到根目录，然后运行叫做*Exuberant Ctags*的工具：

```
find . -iname "*.py" -print | etags -
```

This creates a file called TAGS that contains the location inforation, where every syntactic unit such as classes and functions are defined. In Emacs, you can find the definition of the tag, where your cursor or point as it called in acs is at using the `M-.` command.

While using a tag file is extreely fast for large code bases, it is quite basic and is not aware of a virtual environent where ost definitions ight be located. n excellent alternative is to use the elpy package in acs. It can be configured to detect a virtual environent. uping to a definition of a syntactic eleent is using the same M-. coand. However, the search is not restricted to the tag file. o, you can even jup to a class definition within the Django source code sealessly.

Understanding the code base

It is quite rare to find legacy code with good documentation. Even if you do, the documentation might be out of sync with the code in subtle ways that can lead to further issues. Often, the best guide to understand the application's functionality is the executable test cases and the code itself.

The official Django docuentation has been organied by versions at https://docs. djangoproject.com. On any page, you can quickly switch to the corresponding page in the previous versions of Django with a selector on the bottom right-hand section of the page:

图片：略

In the same way, documentation for any Django package hosted on readthedocs. org can also be traced back to its previous versions. For example, you can select the documentation of django-braces all the way back to v1.0.0 by clicking on the selector on the bottom left-hand section of the page:

图片：略

Creating the big picture 描绘宏伟蓝图

Most people find it easier to understand an application if you show them a high-level diagram. While this is ideally created by someone who understands the workings of the application, there are tools that can create very helpful high-level depiction of a Django application.

很多人发现如果你对他们展示一个高级图表，那么他们会觉得更容易理解一个应用。而这个图表，理论上是由那些理解应用工作流程的人所创建，有很多工具可以创建非常富有帮助的对Django应用的高级描述。

A graphical overview of all models in your apps can be generated by the graph_models management command, which is provided by the django-command-extensions package. As shown in the following diagram, the model classes and their relationships can be understood at a glance:

应用里的全部模型的图形化的概览都可以通过管理命令graph_models生成，它通过包django-command-extensions实现。如下图所示，模型类以及这些模型之间的关系都可以一目了然：

图片：略

Model classes used in the SuperBook project connected by arrows indicating their relationships

This visualization is actually created using PyGraphviz. This can get really large for projects of even medium complexity. Hence, it might be easier if the applications are logically grouped and visualized separately.

实际上，可视化是使用PyGraphviz创建的。这张图可以变得很大，即使面对的时中型的复杂项目。因此，如果应用在逻辑上组织一起，而在视觉上独立的，那么也能够让人们的理解轻松些。

> **PyGraphviz Installation and Usage**
>
> If you find the installation of Pyraphvi challenging, then don't worry, you are not alone. Recently, I faced numerous issues while installing on Ubuntu, starting from Python 3 incompatibility to incomplete documentation. To save your time, I have listed the steps that worked for me to reach a working setup.
>
> On Ubuntu, you will need the following packages installed to install PyGraphviz:

```
$ sudo apt-get install python3.4-dev graphviz libgraphviz-dev pkg-config
```

> Now activate your virtual environment and run pip to install the development version of PyGraphviz directly from GitHub, which supports Python 3:

```
$ pip install git+http://github.com/pygraphviz/pygraphviz.git#egg=pygraphviz
```

> Next, install django-extensions and add it to your INSTALLED_ APPS. Now, you are all set.
>
> Here is a saple usage to create a raphi dot file for just two apps and to convert it to a PNG image for viewing:

```
$ python manage.py graph_models app1 app2 > models.dot
$ dot -Tpng models.dot -o models.png
```

增量变更还是完全重写？

Often, you would be handed over legacy code by the application owners in the earnest hope that most of it can be used right away or after a couple of minor tweaks. However, reading and understanding a huge and often outdated code base is not an easy job. Unsurprisingly, most programmers prefer to work on greenfield developent.

In the best case, the legacy code ought to be easily testable, well documented, and exible to work in odern environents so that you can start aking incremental changes in no time. In the worst case, you might recommend discarding the existing code and go for a full rewrite. Or, as it is commonly decided, the short-term approach would be to keep making incremental changes, and a parallel long-term effort might be underway for a complete reimplementation.

A general rule of thumb to follow while taking such decisions is—if the cost of rewriting the application and maintaining the application is lower than the cost of maintaining the old application over time, then it is recommended to go for a rewrite. Care must be taken to account for all the factors, such as time taken to get new programmers up to speed, the cost of maintaining outdated hardware, and so on.

Sometimes, the complexity of the application domain becomes a huge barrier against a rewrite, since a lot of knowledge learnt in the process of building the older code gets lost. Often, this dependency on the legacy code is a sign of poor design in the application like failing to externalize the business rules from the application logic.

The worst form of a rewrite you can probably undertake is a conversion, or a mechanical translation from one language to another without taking any advantage of the existing best practices. In other words, you lost the opportunity to modernize the code base by removing years of cruft.

Code should be seen as a liability not an asset. As counter-intuitive as it might sound, if you can achieve your business goals with a lesser amount of code, you have dramatically increased your productivity. Having less code to test, debug, and maintain can not only reduce ongoing costs but also make your organization ore agile and exible to change.

> Code is a liability not an asset. Less code is more maintainable.

Irrespective of whether you are adding features or trimming your code, you must not touch your working legacy code without tests in place.

做出任何的改变之前都应该做测试

In the book Working Effectively with Legacy Code, Michael Feathers defines legacy code as, simply, code without tests. He elaborates that with tests one can easily odify the behavior of the code quickly and verifiably. In the absence of tests, it is impossible to gauge if the change made the code better or worse.

ften, we do not know enough about legacy code to confidently write a test. Michael recommends writing tests that preserve and document the existing behavior, which are called characterization tests.

Unlike the usual approach of writing tests, while writing a characterization test, you will first write a failing test with a duy output, say X, because you don't know what to expect. When the test harness fails with an error, such as "Expected output X but got Y", then you will change your test to expect Y. So, now the test will pass, and it becomes a record of the code's existing behavior.

Note that we might record buggy behavior as well. After all, this is unfamiliar code. Nevertheless, writing such tests are necessary before we start changing the code. Later, when we know the specifications and code better, we can fix these bugs and update our tests (not necessarily in that order).

编写测试的具体步骤

Writing tests before changing the code is similar to erecting scaffoldings before the restoration of an old building. It provides a structural framework that helps you confidently undertake repairs.

You ight want to approach this process in a stepwise anner as follows:

1. Identify the area you need to make changes to. Write characterization tests focusing on this area until you have satisfactorily captured its behavior.
2. Look at the changes you need to ake and write specific test cases for those. Prefer smaller unit tests to larger and slower integration tests.
3. Introduce incremental changes and test in lockstep. If tests break, then try to analyze whether it was expected. Don't be afraid to break even the characterization tests if that behavior is something that was intended to change.

If you have a good set of tests around your code, then you can quickly find the effect of changing your code.

换句话来说，如果你决定通过丢掉自己的代码而不是数据来重写，那么Django对于此事是颇有帮助的。

旧版本的数据库

There is an entire section on legacy databases in Django documentation and rightly so, as you will run into them many times. Data is more important than code, and databases are the repositories of data in most enterprises.

You can odernie a legacy application written in other languages or frameworks by importing their database structure into Django. As an immediate advantage, you can use the Django admin interface to view and change your legacy data.

Django makes this easy with the inspectdb management command, which looks as follows:

```
$ python manage.py inspectdb > models.py
```

当你的设置文件使用旧版本的数据库配置过了，这个命令可以自动地生成应用到模型文件的Python代码。

如果你正在把该方法集成到就旧版本数据库，这里给你一些最佳实践建议：

- 预先了解Django ORM的限制。目前，多个列（合成）主键和非关系型数据库是不支持的。
- 不要要记手动清理生成的模型，例如，移除Django自动创建的ID冗余字段。
- 外键关系可能必须手工定义。在某些数据库中自动生成的模型会包含使用 `_id` 作为前缀的整数字段。
- 将模型组织到独立到应用中。之后，在对应到文件夹中就可以轻松的添加视图，表单和测试了。
- 记住在旧版本的数据库中运行迁移命令将创建Django的管理表（ `django_*` 和 `auth_*` ）。

理想的情况中，自动创建的模型会立即运行起来的，不过在实际情况中，它的运行带来的是很多的尝试和错误。有时候，Django推断的数据类型并不合乎你的期望。另外的情况是，你想要对模型添加 `unique_together` 这样对元信息。

Eventually, you should be able to see all the data that was locked inside that aging PHP application in your familiar Django admin interface. I am sure this will bring a smile to your face.

总结

In this chapter, we looked at various techniques to understand legacy code. Reading code is often an underrated skill. But rather than reinventing the wheel, we need to judiciously reuse good working code whenever possible. In this chapter and the rest of the book, we emphasize the importance of writing test cases as an integral part of coding.

本章，我们浏览了多种技术以理解旧版本的代码。阅读代码是一个经常被低估的技能。不过，相比较于重复发明轮子，我们需要决断重复使用。本书的剩下章节，我们强调的是编写测试案例作为代码完整性的一部分。

In the next chapter, we will talk about writing test cases and the often frustrating task of debugging that follows.

下一章，我们要谈论编写测试用例，以及接下来的经常让人沮丧的调试任务。

第九章 测试与调试

本章，我们将讨论以下话题：

- Test-driven development
- Dos and don'ts of writing tests
- Mocking
- Debugging
- Logging

每个程序员都至少考虑过跳过编写测试。Django中默认的app布局拥有一个包含注释的tests.py模块。它也是测试所需的一个提示器。不过，我们经常希望跳过它。

Django中写测试和写代码很相似。实际上，它就是代码。因此，编写测试的过程看起来像是编写了两次（或者更多次）代码。有时候，我们承受了太多的压力，当我们花了很多时间尝试着去让代码正常工作时，虽然这样做看上去很荒唐。

However, eventually, it is pointless to skip tests if you ever want anyone else to use your code. Imagine that you invented an electric razor and tried to sell it to your friend saying that it worked well for you, but you haven't tested it properly. Being a good friend of yours he or she might agree, but imagine the horror if you told this to a stranger.

为什么要写测试

软件中的测试是为了检查软件本身是否按照人们所期望的那样正常运行。假如不对软件进行测试，你或许自认为自己写的代码可以正常运行，不过你也没有办法证明软件可以正常运行。

此外，重要的是要记得在Python中省略掉单元测试是比较危险的，因为Python存在自然——鸭子类型。跟Haskell这样的语言不同，类型检查在编译时并不能够严格地强制执行。单元测试在运行时得到执行（虽然是独立执行），这也是Python开发的基础。

编写测试是你会体会到什么是谦逊。测试会指出你的错误，而且你也得到了一个提前修正错误到机会。实际上，是有一些人愿意在编写代码之前去写测试到。

以测试驱动的开发

Test-driven development (TDD) is a for of software developent where you first write the test, run the test which would fail first, and then write the iniu code needed to make the test pass. This might sound counter-intuitive. Why do we need to write tests when we know that we have not written any code and we are certain that it will fail because of that?

However, look again. We do eventually write the code that erely satisfies these tests. This eans that these tests are not ordinary tests, they are ore like specifications. They tell you what to expect. These tests or specifications will directly coe fro your client's user stories. You are writing just enough code to ake it work.

The process of testdriven developent has any siilarities to the scientific ethod, which is the basis of odern science. In the scientific ethod, it is iportant to frae the hypothesis first, gather data, and then conduct experients that are repeatable and verifiable to prove or disprove your hypothesis.

My recommendation would be to try TDD once you are comfortable writing tests for your projects. Beginners ight find it difficult to frae a test case that checks how the code should behave. For the same reasons, I wouldn't suggest TDD for exploratory programming.

一个编写测试的例子

There are different kinds of tests. However, at the minimum, a programmers need to know unit tests since they have to be able to write them. Unit testing checks the smallest testable part of an application. Integration testing checks whether these parts work well with each other.

测试存在不同的类型。不过，

The word unit is the key term here. Just test one unit at a time. Let's take a look at a simple example of a test case:

```
# tests.py
   from django.test import TestCase
   from django.core.urlresolvers import resolve
   from .views import HomeView
   class HomePageOpenTestCase(TestCase):
       def test_home_page_resolves(self):
           view = resolve('/')
           self.assertEqual(view.func.__name__,
                            HomeView.as_view().__name__)
```

This is a simple test that checks whether, when a user visits the root of our website's domain, they are correctly taken to the home page view. Like most good tests, it has a long and self-descriptive name. The test simply uses Django's resolve() function to match the view callable mapped to the "/" root location to the known view function by their names.

It is more important to note what is not done in this test. We have not tried to retrieve the HTML contents of the page or check its status code. We have restricted ourselves to test just one unit, that is, the resolve() function, which maps the URL paths to view functions.

Assuming that this test resides in, say, app1 of your project, the test can be run with the following command:

```
$ ./manage.py test app1
  Creating test database for alias 'default'...
  .
  -----------------------------------------------------------------
  Ran 1 test in 0.088s
  OK
  Destroying test database for alias 'default'...
```

This command runs all the tests in the app1 application or package. The default test runner will look for tests in all modules in this package matching the pattern test*.py.

Django now uses the standard unittest module provided by Python rather than bundling its own. You can write a testcase class by subclassing from django.test.TestCase. This class typically has methods with the following naming convention:

• test*: Any method whose name starts with test will be executed as a test method. It takes no parameters and returns no values. Tests will be run in an alphabetical order. • setUp (optional): This method will be run before each test method. It can be used to create common objects or perform other initialization tasks that bring your test case to a known state. • tearDown (optional): This method will be run after a test method, irrespective of whether the test passed or not. Clean-up tasks are usually performed here.

A test case is a way to logically group test methods, all of which test a scenario. When all the test methods pass (that is, do not raise any exception), then the test case is considered passed. If any of them fail, then the test case fails.

断言方法

Each test method usually invokes an assert*() method to check some expected outcoe of the test. In our first exaple, we used assertEqual() to check whether the function name matches with the expected function.

Similar to assertEqual(), the Python 3 unittest library provides more than 32 assert ethods. It is further extended by Django by ore than fraeworkspecific assert ethods. You ust choose the ost appropriate ethod based on the end outcome that you are expecting so that you will get the most helpful error message.

Let's see why by looking at an example testcase that has the following setUp() method:

```
def setUp(self):
       self.l1 = [1, 2]
       self.l2 = [1, 0]
```

Our test is to assert that l1 and l2 are equal (and it should fail, given their values). Let's take a look at several equivalent ways to accomplish this:

表格：略

The first stateent uses Python's built in assert keyword. Notice that it throws the least helpful error. You cannot infer what values or types are in the self.l1 and self.l2 variables. This is primarily the reason why we need to use the assert*() methods.

Next, the exception thrown by assertEqual() very helpfully tells you that you are comparing two lists and even tells you at which position they begin to differ. This is exactly similar to the exception thrown by the more specialized assertListEqual() function. This is because, as the documentation would tell you, if assertEqual() is given two lists for comparison, then it hands it over to assertListEqual().

Despite this, as the last example proves, it is always better to use the ost specific assert* method for your tests. Since the second argument is not a list, the error clearly tells you that a list was expected.

> se the ost specific assert* method in your tests.

Therefore, you need to familiarize yourself with all the assert methods, and choose the ost specific one to evaluate the result you expect. This also applies to when you are checking whether your application does not do things it is not supposed to do, that is, a negative test case. You can check for exceptions or warnings using assertRaises and assertWarns respectively.

编写更好一些的测试

We have already seen that the best test cases test a small unit of code at a time. They also need to be fast. A programmer needs to run tests at least once before every commit to the source control. Even a delay of a few seconds can tempt a programmer to skip running tests (which is not a good thing).

Here are some qualities of a good test case (which is a subjective term, of course) in the form of an easy-to-remember mnemonic "F.I.R.S.T. class test case":

1. Fast: the faster the tests, the more often they are run. Ideally, your tests should complete in a few seconds.
2. Independent: Each test case must be independent of others and can be run in any order.
3. Repeatable: The results must be the same every time a test is run. Ideally, all random and varying factors must be controlled or set to known values before a test is run.
4. Small: Test cases must be as short as possible for speed and ease of understanding.

5. Transparent: Avoid tricky implementations or ambiguous test cases.

Additionally, make sure that your tests are automatic. Eliminate any manual steps, no matter how small. Automated tests are more likely to be a part of your team's workow and easier to use for tooling purposes.

Perhaps, even more important are the don'ts to remember while writing test cases:

• Do not (re)test the framework: Django is well tested. Don't check for URL lookup, template rendering, and other framework-related functionality. • Do not test implementation details: Test the interface and leave the minor implementation details. It makes it easier to refactor this later without breaking the tests. • Test models most, templates least: Templates should have the least business logic, and they change more often. • Avoid HTML output validation: Test views use their context variable's output rather than its HTML-rendered output. • Avoid using the web test client in unit tests: Web test clients invoke several components and are therefore, better suited for integration tests. • Avoid interacting with external systems: Mock them if possible. Database is an exception since test database is in-memory and quite fast.

Of course, you can (and should) break the rules where you have a good reason to just like I did in y first exaple. Itiately, the ore creative you are at writing tests, the earlier you can catch bugs, and the better your application will be.

Mocking

Most real-life projects have various interdependencies between components. While testing one component, the result must not be affected by the behavior of other components. For example, your application might call an external web service that might be unreliable in terms of network connection or slow to respond.

Mock objects imitate such dependencies by having the same interface, but they respond to method calls with canned responses. After using a mock object in a test, you can assert whether a certain method was called and verify that the expected interaction took place.

Take the exaple of the uperHero profile eligibility test entioned in Pattern: Service objects (see Chapter 3, Models). We are going to mock the call to the service object method in a test using the Python 3 unittest.mock library:

```
# profiles/tests.py
  from django.test import TestCase
  from unittest.mock import patch
  from django.contrib.auth.models import User
  class TestSuperHeroCheck(TestCase):
      def test_checks_superhero_service_obj(self):
          with patch("profiles.models.SuperHeroWebAPI") as ws:
              ws.is_hero.return_value = True
              u = User.objects.create_user(username="t")
              r = u.profile.is_superhero()
          ws.is_hero.assert_called_with('t')
          self.assertTrue(r)
```

Here, we are using patch() as a context manager in a with statement. Since the profile odel's is_superhero() method will call the SuperHeroWebAPI.is_hero() class method, we need to mock it inside the models module. We are also hard-coding the return value of this method to be True.

The last two assertions check whether the method was called with the correct arguments and if is_hero() returned True, respectively. Since all methods of SuperHeroWebAPI class have been mocked, both the assertions will pass.

Mock objects come from a family called Test Doubles, which includes stubs, fakes, and so on. Like movie doubles who stand in for real actors, these test doubles are used in place of real objects while testing. While there are no clear lines drawn between them, Mock objects are objects that can test the behavior, and stubs are simply placeholder implementations.

Pattern test fitures and factories

Problem: Testing a component requires the creation of various prerequisite objects before the test. Creating them explicitly in each test method gets repetitive.

Solution: Utilize factories or fixtures to create the test data objects.

问题细节

Before running each test, Django resets the database to its initial state, as it would be after running migrations. Most tests will need the creation of some initial objects to set the state. Rather than creating different initial objects for different scenarios, a common set of initial objects are usually created.

This can quickly get unmanageable in a large test suite. The sheer variety of such initial objects can be hard to read and later understand. This leads to hardtofind bugs in the test data itself!

Being such a common problem, there are several means to reduce the clutter and write clearer test cases.

解决方法细节

The first solution we will take a look at is what is given in the Django docuentation itselftest fixtures. Here, a test fixture is a file that contains a set of data that can be iported into your database to bring it to a known state. Typically, they are YML or files previously exported from the same database when it had some data.

For exaple, consider the following test case, which uses a test fixture:

```
from django.test import TestCase
   class PostTestCase(TestCase):
       fixtures = ['posts']
       def setUp(self):
           # Create additional common objects
           pass
       def test_some_post_functionality(self):
           # By now fixtures and setUp() objects are loaded
           pass
```

Before setUp() gets called in each test case, the specified fixture, posts gets loaded. Roughly speaking, the fixture would be searched for in the fixtures directory with certain known extensions, for example, app/fixtures/posts.json.

However, there are a nuber of probles with fixtures. Fixtures are static snapshots of the database. They are schema-dependent and have to be changed each time your models change. They also might need to be updated when your test-case assertions change. pdating a large fixture file anually, with ultiple related objects, is no joke.

For all these reasons, any consider using fixtures as an antipattern. It is recommended that you use factories instead. A factory class creates objects of a particular class that can be used in tests. It is a DRY way of creating initial test objects.

Let's use a model's objects.create method to create a simple factory:

```
from django.test import TestCase
   from .models import Post
   class PostFactory:
       def make_post(self):
           return Post.objects.create(message="")
   class PostTestCase(TestCase):
       def setUp(self):
           self.blank_message = PostFactory().makePost()
       def test_some_post_functionality(self):
           pass
```

opared to using fixtures, the initial object creation and the test cases are all in one place. Fixtures load static data as is into the database without calling odeldefined save() methods. Since factory objects are dynamically generated, they are more likely to run through your application's custom validations.

However, there is a lot of boilerplate in writing such factory classes yourself. The factory_boy package, based on thoughtbot's factory_girl, provides a declarative syntax for creating object factories.

Rewriting the previous code to use factory_boy, we get the following result:

```
import factory
   from django.test import TestCase
   from .models import Post
   class PostFactory(factory.Factory):
       class Meta:
           model = Post
       message = ""
   class PostTestCase(TestCase):
       def setUp(self):
           self.blank_message = PostFactory.create()
           self.silly_message = PostFactory.create(message="silly")
       def test_post_title_was_set(self):
           self.assertEqual(self.blank_message.message, "")
           self.assertEqual(self.silly_message.message, "silly")
```

Notice how clear the `factory` class becomes when written in a declarative fashion. The attribute's values do not have to be static. You can have sequential, rando, or computed attribute values. If you prefer to have more realistic placeholder data such as US addresses, then use the `django-faker` package.

In conclusion, I would recommend factories, especially factory_boy, for most projects that need initial test objects. One ight still want to use fixtures for static data, such as lists of countries or t-shirt sizes, since they would rarely change.

> **Dire Predictions**
>
> After the announcement of the impossible deadline, the entire team seemed to be suddenly out of time. They went from 4-week scrum sprints to 1-week sprints. Steve wiped every meeting off their calendars except "today's 30-minute catch-up with Steve." He preferred to have a one-on-one discussion if he needed to talk to someone at their desk.
>
> At Madam O's insistence, the 30-minute meetings were held at a sound proof hall 20 levels below the S.H.I.M. headquarters. On Monday, the team stood around a large circular table with a gray metallic surface like the rest of the room. Steve stood awkwardly in front of it and made a stiff waving gesture with an open palm.

> Even though everyone had seen the holographs come alive before, it never failed to amaze them each time. The disc almost segmented itself into hundreds of metallic squares and rose like miniature skyscrapers in a futuristic model city. It took them a second to realize that they were looking at a 3D bar chart.
>
> "Our burn-down chart seems to be showing signs of slowing down. I am guessing it is the outcome of our recent user tests, which is a good thing. But..." Steve's face seemed to show the strain of trying to stie a sneee. He gingerly icked his forefinger upwards in the air and the chart smoothly extended to the right.
>
> "At this rate, projections indicate that we will miss the go-live by several days, at best. I did a bit of analysis and found several critical bugs late in our development. We can save a lot of time and effort if we can catch them early. I want to put your heads together and come up with some i..."
>
> Steve clasped his mouth and let out a loud sneeze. The holograph interpreted this as a sign to zoom into a particularly uninteresting part of the graph. Steve cursed under his breath and turned it off. He borrowed a napkin and started noting down everyone's suggestions with an ordinary pen.
>
> One of the suggestions that Steve liked most was a coding checklist listing the most common bugs, such as forgetting to apply migrations. He also liked the idea of involving users earlier in the development process for feedback. He also noted down some unusual ideas, such as a Twitter handle for tweeting the status of the continuous integration server.
>
> At the close of the meeting, Steve noticed that Evan was missing. Where is van? he asked. o idea, said Brad looking confused, "he was here a minute ago."
>
> # Learning more about testing
>
> Django's default test runner has improved a lot over the years. However, test runners such as py.test and nose are still superior in terms of functionality. They make your tests easier to write and run. Even better, they are compatible with your existing test cases.

You ight also be interested in knowing what percentage of your code is covered by tests. This is called Code coverage and coverage.py is a very popular tool for finding this out.

Most projects today tend to use a lot of JavaScript functionality. Writing tests for them usually require a browser-like environment for execution. Selenium is a great browser automation tool for executing such tests.

While a detailed treatment of testing in Django is outside the scope of this book, I would strongly recommend that you learn more about it.

If nothing else, the two main takeaways I wanted to convey through this section are first, write tests, and second, once you are confident at writing the, practice TDD.

Debugging

Despite the most rigorous testing, the sad reality is, we still have to deal with bugs. Django tries its best to be as helpful as possible while reporting an error to help you in debugging. However, it takes a lot of skill to identify the root cause of the problem.

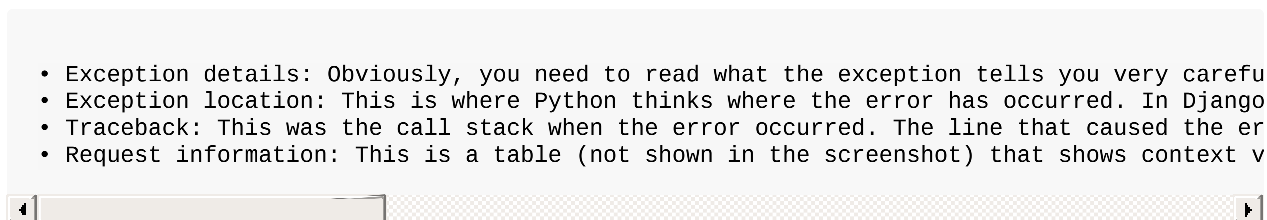
Thankfully, with the right set of tools and techniques, we can not only identify the bugs but also gain great insight into the runtime behavior of your code. Let's take a look at some of these tools.

Django的调试页面

If you have encountered any exception in development, that is, when DEBUG=True, then you would have already seen an error page similar to the following screenshot:

图片：略

Since it comes up so frequently, most developers tend to miss the wealth of information in this page. Here are some places to take a look at:



```
• Exception details: Obviously, you need to read what the exception tells you very carefu
• Exception location: This is where Python thinks where the error has occurred. In Django
• Traceback: This was the call stack when the error occurred. The line that caused the er
• Request information: This is a table (not shown in the screenshot) that shows context v
```

更友好的测试页面

Often, you may wish for more interactivity in the default Django error page. The django-extensions package ships with the fantastic Werkzeug debugger that provides exactly this feature. In the following screenshot of the same exception, notice a fully interactive Python interpreter available at each level of the call stack:

图片：略

To enable this, in addition to adding django_extensions to your INSTALLED_APPS, you will need to run your test server as follows:

```
$ python manage.py runserver_plus
```

Despite the reduced debugging inforation, I find the Werkeug debugger to be more useful than the default error page.

print函数

Sprinkling print() functions all over the code for debugging might sound primitive, but it has been the preferred technique for many programmers.

Typically, the print() functions are added before the line where the exception has occurred. It can be used to print the state of variables in various lines leading to the exception. You can trace the execution path by printing soething when a certain line is reached.

In development, the print output usually appears in the console window where the test server is running. Whereas in production, these print outputs might end up in your server log file where they would add a runtie overhead.

In any case, it is not a good debugging technique to use in production. Even if you do, the print functions that are added for debugging should be removed from being committed to your source control.

日志

The ain reason for including the previous section was to sayYou should replace the print() functions with calls to logging functions in Python's logging module. Logging has several advantages over printing: it has a timestamp, a clearly marked level of urgency (for example, INFO, DEBUG), and you don't have to remove them from your code later.

Logging is fundamental to professional web development. Several applications in your production stack, like web servers and databases, already use logs. Debugging might take you to all these logs to retrace the events that lead to a bug. It is only appropriate that your application follows the same best practice and adopts logging for errors, warnings, and informational messages.

Unlike the common perception, using a logger does not involve too much work. Sure, the setup is slightly involved but it is merely a one-time effort for your entire project. Even more, most project templates (for example, the edge template) already do this for you.

nce you have configured the LOGGING variable in settings.py, adding a logger to your existing code is quite easy, as shown here:

```
# views.py
    import logging
    logger = logging.getLogger(__name__)
    def complicated_view():
        logger.debug("Entered the complicated_view()!")
```

The logging module provides various levels of logged messages so that you can easily filter out less urgent essages. The log output can be also formatted in various ways and routed to any places, such as standard output or log files. Read the documentation of Python's logging module to learn more.

Django调试工具

The Django Debug Toolbar is an indispensable tool not just for debugging but also for tracking detailed information about each request and response. Rather than appearing only during exceptions, the toolbar is always present in your rendered page.

Initially, it appears as a clickable graphic on the right-hand side of your browser window. On clicking, a toolbar appears as a dark semi-transparent sidebar with several headers:

图片：略

ach header is filled with detailed inforation about the page fro the nuber of SQL queries executed to the templates that we use to render the page. Since the toolbar disappears when DEBUG is set to False, it is pretty much restricted to being a development tool.

The Python debugger pdb While debugging, you might need to stop a Django application in the middle of execution to examine its state. A simple way to achieve this is to raise an exception with a simple assert False line in the required place. What if you wanted to continue the execution step by step fro that line? This is possible with the use of an interactive debugger such as Python's pdb. Simply insert the following line wherever you want the execution to stop and switch to pdb:

```
import pdb; pdb.set_trace()
```

Once you enter pdb, you will see a command-line interface in your console window with a (Pdb) prompt. At the same time, your browser window will not display anything as the request has not finished processing.

The pdb command-line interface is extremely powerful. It allows you to go through the code line by line, examine the variables by printing them, or execute arbitrary code that can even change the running state. The interface is quite similar to GDB, the GNU debugger.

其他的调试器

There are several drop-in replacements for pdb. They usually have a better interface. Some of the console-based debuggers are as follows:

• ipdb: Like IPython, this has autocomplete, syntax-colored code, and so on. • pudb: Like old Turbo C IDEs, this shows the code and variables side by side. • IPython: This is not a debugger. You can get a full IPython shell anywhere in your code by adding the from IPython import embed; embed()line.

PuDB is my preferred replacement for pdb. It is so intuitive that even beginners can easily use this interface. Like pdb, just insert the following code to break the execution of the program:

```
import pudb; pudb.set_trace()
```

When this line is executed, a full-screen debugger is launched, as shown here:

图片：略

Press the ? key to get help on the complete list of keys that you can use.

此外，还存在多种图形化的调试器，有些是独立使用的，比如winpdb和其他的集成到IDE中的调试器，比如PyCharm,PyDev,和Komodo。我们会推荐你多去尝试几个调试器，直道你发现那个适合你的工具。

调试Django模板

在项目中模板会存在非常复杂的逻辑。在创建模板时细微的bug会引发难以发现的不过。我们需要在`settings.py`中设置TEMPLATE_DEBUG为True，这样Django就可以更好在模板中出现错误时显示错误页面。

有多种简陋的方法来调试模板，比如插入想要的变量，，或者是你想要像这样使用内建的调试标签，拉取所有的变量（在一个很方便的，可点击的文本区域内部）：

```
<textarea onclick="this.focus();this.select()" style="width: 100%;">
     {% filter force_escape %}
{% debug %}
     {% endfilter %}
   </textarea>
```

更好的选择是使用之前提到的Django调试工具。它不仅能够告诉你上下文变量中的值，而且还显示了模板中的继承树。

不过，你要是想要在模板中部暂停以检查状态（即，在循环内部），调试将会是此类情况的最佳选择。实际上，你可以使用之前提到Python解释器中的任何一个来为你的模板使用自定义模板标签。

下面是一个此类模板标签的简单实现。在模板标签的包目录内创建以下文件：

```
# templatetags/debug.py
   import pudb as dbg              # Change to any *db
   from django.template import Library, Node
register = Library()
class PdbNode(Node):
    def render(self, context):
        dbg.set_trace()
return ''
@register.tag
def pdb(parser, token):
# Debugger will stop here
return PdbNode()
```

在模板中，载入模板标签库，在需要执行暂停的地方插入pdg标签，以及输入调试器：

```
{% load debug %}
   {% for item in items %}
       {# Some place you want to break #}
       {% pdb %}
{% endfor %}
```

在调试器内部，你可以验证任何事情，使用上下文字典引入上下文变量：

```
>>> print(context["item"])
Item0
```

如果你需要在调试和内省上使用更多的模板标签，那么我会推荐你试验下 `django-template-debug` 包。

总结

这一章，我们浏览了Django中执行测试的背后动机和概念。我们也发现了在编写测试时所遵循的多种最佳实践。

在调试的小节，我们熟悉了使用多种调试工具和技术发现在Django代码和模板的问题。

在下一章，我们会通过理解多种安全问题，以及如何减少来自各种恶意攻击的威胁，来更进一步靠近生产环境中的代码。

第十章 安全

这一章，我们谈及以下议题：

- 各种web攻击及其对策
- Django对于安全问题的能与不能
- 对Django应用的安全检查

跨站脚本（XSS）

为什么cookie值得如此关注

In this chapter, we will discuss the following topics:
在这一章，我们将讨论以下话题：

• Picking a web stack 挑选一个web服务
• Hosting approaches 托管方法
• Deployment tools 发布工具
• Monitoring 监控
• Performance tips 性能建议

So, you have developed and tested a fully functional web application in Django. Deploying this application can involve a diverse set of activities from choosing your hosting provider to performing installations. Even more challenging could be the tasks of maintaining a production site working without interruptions and handling unexpected bursts in traffic.

你已经使用Django开发并测试了完整功能的web应用。而部署这个应用就涉及到了选择你的主机托管服务商执行安装的一组各种不同的行为。甚至是更具挑战的维护生产环境中正在使用的网站，而不用中断它，并处理未被预料到到突发流量。

The discipline of system administration is vast. Hence, this chapter will cover a lot of ground. However, given the limited space, we will attempt to familiarize you with the various aspects of building a production environment.

系统管理员的规定是很多的。因此，本章覆盖的内容也很多。篇幅虽然所限，但是我们会试着让你尽可能多的熟悉构建生成环境的方方面面。

生成环境

Although, most of us intuitively understand what a production environment is, it is worthwhile to clarify what it really means. A production environment is simply one where end users use your application. It should be available, resilient, secure, responsive, and must have abundant capacity for current (and future) needs.
尽管，我们都大多数的人从直觉上都能够理解生产环境是个什么东西，但是在这里，值得我们花点时间来澄清生产环境的真正的意思。生产环境是一个多简单的终端用户可以使用你所开发的应用的地方。它应该是可用的、可还原的、安全的、灵活响应的、而且必须拥有当前（和未来）充足的扩展能力。

Unlike a development environment, the chance of real business damage due to any issues in a production environment is high. Hence, before moving to production, the code is moved to various testing and acceptance environments in order to get rid of as many bugs as possible. For easy traceability, every change made to the production environment must be tracked, documented, and made accessible to everyone in the team.

不同于开发环境，在生产环境中对现实业务的破坏所引发的任何问题都可能要付出非常高昂都代价。因此，在迁移到生产环境中之前，代码应该接受多种测试，和认可的环境中以尽可能多多去除bug。简单来说，每个应用到生产环境中的变更都必须追踪、纪录、而且要然团队中任何人都能够理解。

As an upshot, there must be no development performed directly on the production environment. In fact, there is no need to install development tools, such as a compiler or debugger in production. The presence of any additional software increases the attack surface of your site and could pose a security risk.

Most web applications are deployed on sites with extremely low downtime, say, large data centers running 24/7/365. By designing for failure, even if an internal component fails, there is enough redundancy to prevent the entire system crashing. This concept of avoiding a single point of failure (SPOF) can be applied at every level—hardware or software.

Hence, it is crucial which collection of software you choose to run in your production environment.

选择**web**服务

So far, we have not discussed the stack on which your application will be running on. Even though we are talking about it at the very end, it is best not to postpone such decisions to the later stages of the application lifecycle. Ideally, your development environment must be as close as possible to the production environment to avoid the "but it works on my machine" argument.

By a web stack, we refer to the set of technologies that are used to build a web application. It is usually depicted as a series of components, such as OS, database, and web server, all piled on top of one another. Hence, it is referred to as a stack.

We will mainly focus on open source solutions here because they are widely used. However, various commercial applications can also be used if they are more suited to your needs.

栈的组件

A production Django web stack is built using several kinds of application (or layers, depending on your terminology). While constructing your web stack, some of the choices you might need to make are as follows:

生产环境中的Django web栈使用的是多种应用（或者层，视使用的术语不同而不同）。当构建web栈时，如下是你可能遇到当选择：

- Which and distribution? For exaple Debian, Red Hat, or penBD.
- Which WI server? For exaple unicorn, uWI.

- Which web server? For exaple pache, ginx.
- Which database? For exaple PostgreL, MyL, or Redis.
- Which caching syste? For exaple Mecached, Redis.
- Which process control and onitoring syste? For exaple pstart, Systemd, or Supervisord.
- How to store static edia? For exaple aon , loudFront.

There could be several more, and these choices are not mutually exclusive either. Some use several of these applications in tandem. For example, username availability might be looked up on Redis, while the primary database might be PostgreSQL.

There is no 'one sie fits all' answer when it coes to selecting your stack. Different components have different strengths and weaknesses. Choose them only after careful consideration and testing. For instance, you might have heard that Nginx is a popular choice for a web server, but you might actually need Apache's rich ecosystem of modules or options.

Sometimes, the selection of the stack is based on various non-technical reasons. Your organiation ight have standardied on a particular operating syste, say, Debian for all its servers. Or your cloud hosting provider might support only a limited set of stacks.

Hence, how you choose to host your Django application is one of the key factors in determining your production setup.

托管

When it comes to hosting, you need to make sure whether to go for a hosting platform such as Heroku or not. If you do not know much about managing a server or do not have anyone with that knowledge in your team, then a hosting platform is a convenient option.

平台即服务

Platform as a service A Platform as a ervice Paa is defined as a cloud service where the solution stack is already provided and managed for you. Popular platforms for Django hosting include Heroku, PythonAnywhere, and Google App Engine.

In most cases, deploying a Django application should be as simple as selecting the services or coponents of your stack and pushing out your source code. You do not have to perform any system administration or setup yourself. The platform is entirely managed.

Like most cloud services, the infrastructure can also scale on demand. If you need an additional database server or more RAM on a server, it can be easily provisioned from a web interface or the command line. The pricing is primarily based on your usage.

The bottom line with such hosting platforms is that they are very easy to set up and ideal for smaller projects. They tend to be more expensive as your user base grows.

Another downside is that your application might get tied to a platform or becoe difficult to port. For instance, oogle pp ngine is used to support only a non-relational database, which means you need to use django-nonrel, a fork of Django. This limitation is now somewhat mitigated with Google Cloud SQL.

虚拟私有服务器

A virtual private server (VPS) is a virtual machine hosted in a shared environment. From the developer's perspective, it would seem like a dedicated machine (hence, the word private preloaded with an operating syste. You will need to install and set up the entire stack yourself, though many VPS providers such as WebFaction and DigitalOcean offer easier Django setups.

If you are a beginner and can spare some time, I highly recommend this approach. You would be given root access, and you can build the entire stack yourself. You will not only understand how various pieces of the stack come together but also have full control in finetuning each individual coponent.

Compared to a PaaS, a VPS might work out to be more value for money, especially for hightraffic sites. You ight be able to run several sites fro the same server as well.

另外托管主机的方法

Other hosting approaches Even though hosting on a platform or VPS are by far the two most popular hosting options, there are plenty of other options. If you are interested in maximizing performance, you can opt for a bare metal server with colocation from providers, such as Rackspace.

On the lighter end of the hosting spectrum, you can save the cost by hosting multiple applications within Docker containers. Docker is a tool to package your application and dependencies in a virtual container. Compared to traditional virtual machines, a Docker container starts up faster and has minimal overheads (since there is no bundled operating system or hypervisor).

Docker is ideal for hosting micro services-based applications. It is becoming as ubiquitous as virtualization with almost every PaaS and VPS provider supporting them. It is also a great development platform since Docker containers encapsulate the entire application state and can be directly deployed to production.

开发工具

Once you have zeroed in on your hosting solution, there could be several steps in your deployment process, from running regression tests to spawning background services.

The key to a successful deployment process is automation. Since deploying applications involve a series of welldefined steps, it can be rightly approached as a programming problem. Once you have an automated deployment in place, you do not have to worry about deployments for fear of missing a step.

In fact, deployments should be painless and as frequent as required. For example, the Facebook team can release code to production up to twice a day. Considering Facebook's enormous user base and code base, this is an impressive feat, yet, it becoes necessary as eergency bug fixes and patches need to be deployed as soon as possible.

A good deployment process is also idempotent. In other words, even if you accidentally run the deployment tool twice, the actions should not be executed twice (or rather it should leave it in the same state).

Let's take a look at some of the popular tools for deploying Django applications.

Fabric

Fabric is favored among Python web developers for its simplicity and ease of use. It expects a file naed fabfile.py that defines all the actions for deployent or otherwise) in your project. Each of these actions can be a local or remote shell command. The remote host is connected via SSH.

The key strength of Fabric is its ability to run commands on a set of remote hosts. For instance, you can define a web group that contains the hostnames of all web servers in production. You can run a Fabric action only against these web servers by specifying the web group name on the command line.

To illustrate the tasks involved in deploying a site using Fabric, let's take a look at a typical deployment scenario.

典型的部署步骤

Imagine that you have a medium-sized web application deployed on a single web server. Git has been chosen as the version control and collaboration tool. A central repository that is shared with all users has been created in the form of a bare Git tree.

Let's assume that your production server has been fully set up. When you run your Fabric deployment command, say, fab deploy, the following scripted sequence of actions take place:

1. Run all tests locally.
2. Commit all local changes to Git.
3. Push to a remote central Git repository.
4. Resolve erge conicts, if any.
5. ollect the static files , iages.
6. opy the static files to the static file server.
7. At remote host, pull changes from a central Git repository.
8. At remote host, run (database) migrations.
9. At remote host, touch app.wsgi to restart WSGI server.

The entire process is automatic and should be completed in a few seconds. By default, if any step fails, then the deployment gets aborted. Though not explicitly mentioned, there would be checks to ensure that the process is idempotent.

Note that Fabric is not yet compatible with Python 3, though the developers are in the process of porting it. In the meantime, you can run Fabric in a Python 2.x virtual environment or check out similar tools, such as PyInvoke.

配置管理

Managing multiple servers in different states can be hard with Fabric. onfiguration management tools such as Chef, Puppet, or Ansible try to bring a server to a certain desired state.

nlike Fabric, which requires the deployent process to be specified in an iperative anner, these configurationanageent tools are declarative. You just need to define the final state you want the server to be in, and it will figure out how to get there.

For example, if you want to ensure that the Nginx service is running at startup on all your web servers, then you need to define a server state having the ginx service both running and starting on boot. On the other hand, with Fabric, you need to specify the exact steps to install and configure ginx to reach such a state.

ne of the ost iportant advantages of configurationanageent tools is that they are idepotent by default. Your servers can go fro an unknown state to a known state, resulting in easier server configuration anageent and reliable deployent. ong configurationanageent tools, hef and Puppet enjoy wide popularity since they were one of the earliest tools in this category. However, their roots in Ruby can make them look a bit unfamiliar to the Python programmer. For such folks, we have Salt and Ansible as excellent alternatives.

onfigurationanageent tools have a considerable learning curve copared to simpler tools, such as Fabric. However, they are essential tools for creating reliable production environments and are certainly worth learning.

监控

Even a medium-sized website can be extremely complex. Django might be one of the hundreds of applications and services running and interacting with each other. In the same way that the heart beat and other vital signs can be constantly monitored to assess the health of the human body, so are various metrics collected, analyzed, and presented in most production systems.

While logging keeps track of various events, such as arrival of a web request or an exception, monitoring usually refers to collecting key information periodically, such as memory utilization or network latency. However, differences get blurred at application level, such as, while monitoring database query performance, which might very well be collected from logs.

Monitoring also helps with the early detection of problems. Unusual patterns, such as spikes or a gradually increasing load, can be signs of bigger underlying problems, such as a memory leak. A good monitoring system can alert site owners of problems before they happen. Monitoring tools usually need a backend service (sometimes called agents) to collect the statistics, and a frontend service to display dashboards or generate reports. Popular data collection backends include StatsD and Monit. This data can be passed to frontend tools, such as Graphite.

There are several hosted monitoring tools, such as New Relic and Status.io, which are easier to set up and use.

Measuring performance is another important role of monitoring. As we will soon see, any proposed optimization must be carefully measured and monitored before getting implemented.

性能

Performance is a feature. Studies show how slow sites have an adverse effect on users, and therefore, revenue. For instance, tests at Amazon in 2007 revealed that for every 100 ms increase in load time of amazon.com, the sales decreased by 1 percent.

Reassuringly, several high-performance web applications such as Disqus and Instagram have been built on Django. At Disqus, in 2013, they could handle 1.5 million concurrently connected users, 45,000 new connections per second, 165,000 messages/second, with less than 0.2 seconds latency end-to-end.

The key to iproving perforance is finding where the bottlenecks are. Rather than relying on guesswork, it is always recoended that you easure and profile your application to identify these performance bottlenecks. As Lord Kelvin would say:

```
If you can't measure it, you can't improve it.
```

In most web applications, the bottlenecks are likely to be at the browser or the database end rather than within Django. However, to the user, the entire application needs to be responsive.

Let's take a look at some of the ways to improve the performance of a Django application. Due to widely differing techniques, the tips are split into two parts: frontend and backend.

前端性能

Django programmers might quickly overlook frontend performance because it deals with understanding how the client-side, usually a browser, works. However, to quote Steve Souders' study of Alexa-ranked top 10 websites:

```
80-90% of the end-user response time is spent on the frontend. Start there.
```

A good starting point for frontend optimization would be to check your site with oogle Page peed or Yahoo Ylow coonly used as browser plugins. These tools will rate your site and recommend various best practices, such as minimizing the number of HTTP requests or gzipping the content.

As a best practice, your static assets, such as images, style sheets, and JavaScript files ust not be served through Django. Rather a static file server, cloud storages such as Amazon S3 or a content delivery network (CDN) should serve them for better performance.

Even then, Django can help you improve frontend performance in a number of ways:

```
• Cache infinitely with CachedStaticFilesStorage: The fastest way to load static assets i
CachedStaticFilesStorage solves this elegantly by appending the asset's MD5 hash to its f

To use this, set the STATICFILES_STORAGE to CachedStaticFilesStorage or, if you have a cu

• Use a static asset manager: An asset manager can preprocess your static assets to minif
```

后端性能

The scope of backend performance improvements covers your entire server-side web stack, including database queries, template rendering, caching, and background jobs. You will want to extract the highest perforance fro the, since it is entirely within your control.

For quick and easy profiling needs, django-debug-toolbar is quite handy. We can also use Python profiling tools, such as the hotshot module for detailed analysis. In Django, you can use one of the several profiling iddleware snippets to display the output of hotshot in the browser.

recent liveprofiling solution is django-silk. It stores all the requests and responses in the configured database, allowing aggregated analysis over an entire user session, say, to find the worstperforing views. It can also profile any piece of Python code by adding a decorator.

As before, we will take a look at some of the ways to improve backend performance. However, considering they are vast topics in themselves, they have been grouped into sections. Many of these have already been covered in the previous chapters but have been summarized here for easy reference.

模板

As the documentation suggests, you should enable the cached template loader in production. This avoids the overhead of reparsing and recompiling the templates each tie it needs to be rendered. The cached teplate is copiled the first tie it is needed and then stored in memory. Subsequent requests for the same template are served from memory.

If you find that another teplating language such as inja renders your page significantly faster, then it is quite easy to replace the builtin Django teplate language. There are several libraries that can integrate Django and Jinja2, such as django-jinja. Django 1.8 is expected to support multiple templating engines out of the box.

数据库

Sometimes, the Django RM can generate inefficient L code. There are several optimization patterns to improve this:

```
• Reduce database hits with select_related: If you are using a OneToOneField or a Foreign
• Reduce database hits with prefetch_related: For accessing a ManyToManyField method or,
• Fetch only needed fields with values or values_list You can save tie and memory usage b
• Denormalize models: Selective denormalization improves performance by reducing joins at
• Add an Index: If a non-primary key gets searched a lot in your queries, consider settin
• Create, update, and delete multiple rows at once: Multiple objects can be operated upon
```

As a last resort, you can always finetune the raw L stateents using proven database performance expertise. However, maintaining the SQL code can be painful over time.

缓存

Any computation that takes time can take advantage of caching and return precomputed results faster. However, the problem is stale data or, often, quoted as one of the hardest things in computer science, cache invalidation. This is coonly spotted when, despite refreshing the page, a YouTube video's view count doesn't change.

Django has a exible cache system that allows you to cache anything from a template fragment to an entire site. It allows a variety of pluggable backends such as filebased or databased backed storage.

Most production systems use a memory-based caching system such as Redis or Memcached. This is purely because volatile memory is many orders of magnitude faster than disk-based storage.

Such cache stores are ideal for storing frequently used but ephemeral data, like user sessions.

缓存session后端

By default, Django stores its user session in the database. This usually gets retrieved for every request. To improve performance, the session data can be stored in memory by changing the SESSION_ENGINE setting. For instance, add the following in settings.py to store the session data in your cache:

```
SESSION_ENGINE = "django.contrib.sessions.backends.cache"
```

Since some cache storages can evict stale data leading to the loss of session data, it is preferable to use Redis or Memcached as the session store, with memory limits high enough to support the maximum number of active user sessions.

缓存框架

For basic caching strategies, it might be easier to use a caching framework. Two popular ones are django-cache-machine and django-cachalot. They can handle common scenarios, such as automatically caching results of queries to avoid database hits every time you perform a read.

就基本的缓存策略来说，

The simplest of these is Django-cachalot, a successor of Johnny Cache. It requires very little configuration. It is ideal for sites that have multiple reads and infrequent writes (that is, the vast majority of applications), it caches all Django ORM read queries in a consistent manner.

缓存模式

nce your site starts getting heavy traffic, you will need to start exploring several caching strategies throughout your stack. Using Varnish, a caching server that sits between your users and Django, many of your requests might not even hit the Django server.

Varnish can make pages load extremely fast (sometimes, hundreds of times faster than normal). However, if used improperly, it might serve static pages to your users. arnish can be easily configured to recognie dynaic pages or dynaic parts of a page such as a shopping cart.

Russian doll caching, popular in the Rails community, is an interesting template cache-invalidation pattern. Imagine a user's timeline page with a series of posts each containing a nested list of comments. In fact, the entire page can be considered as several nested lists of content. At each level, the rendered template fragment gets cached.

So, if a new comment gets added to a post, only the associated post and timeline caches get invalidated. otice that we first invalidate the cache content directly outside the changed content and move progressively until at the outermost content. The dependencies between models need to be tracked for this pattern to work.

Another common caching pattern is to cache forever. Even after the content changes, the user might get served stale data from the cache. However, an asynchronous job, such as, a elery job, also gets triggered to update the cache. You can also periodically warm the cache at a certain interval to refresh the content. ssentially, a successful caching strategy

identifies the static and dynaic parts of a site. For any sites, the dynaic parts are the userspecific data when you are logged in. If this is separated from the generally available public content, then implementing caching becomes easier.

Don't treat caching as integral to the working of your site. The site must fall back to a slower but working state even if the caching system breaks down.

> 注释
>
> Cranos It was six in the morning and the S.H.I.M. building was surrounded by a grey fog. Somewhere inside, a small conference room had been designated the "War Room." For the last three hours, the SuperBook team had been holed up here diligently executing their pre-go-live plan.
>
> More than 30 users had logged on the IRC chat room #superbookgolive from various parts of the world. The chat log was projected on a giant whiteboard. When the last item was struck off, Evan glanced at Steve. Then, he pressed a key triggering the deployment process.
>
> The room fell silent as the script output kept scrolling off the wall. One error, Steve thought—just one error can potentially set them back by hours. Several seconds later, the command prompt reappeared. It was live! The team erupted in joy. Leaping from their chairs they gave highfives to each other. oe were crying tears of happiness. fter weeks of uncertainty and hard work, it all seemed surreal.
>
> However, the celebrations were short-lived. A loud explosion from above shook the entire building. Steve knew the second breach had begun. He shouted to Evan, "Don't turn on the beacon until you get my message," and sprinted out of the room.

As Steve hurried up the stairway to the rooftop, he heard the sound of footsteps above hi. It was Mada . he opened the door and ung herself in. He could hear her screaming "No!" and a deafening blast shortly after that. By the time he reached the rooftop, he saw Madam O sitting with her back against the wall. She clutched her left arm and was wincing in pain. Steve slowly peered around the wall. At a distance, a tall bald man seemed to be working on something with the help of two robots.

> "He looks like...." Steve broke off, unsure of himself. Yes, it is Hart. Rather I should say he is ranos now. What? Yes, a split personality. onster that laid hidden in Hart's ind for years. I tried to help him control it. Many years back, I thought I had stopped it from ever coming back. However, all this stress took a toll on him. Poor thing, if only I could get near him."
>
> Poor thing indeed—he nearly tried to kill her. Steve took out his mobile and sent out a message to turn on the beacon. He had to improvise. With his hands high in the air and fingers crossed, he stepped out. The two robots immediately aimed directly at him. Cranos motioned them to stop. Well, who do we have here? Mr. uperBook hiself. Did I crash into your launch party, teve?
>
> "It was our launch, Hart."
>
> "Don't call me that," growled Cranos. "That guy was a fool. He wrote the Sentinel code but he never understood its potential. I mean, just look at what Sentinels can do—unravel every cryptographic algorithm known to an. What happens when it enters an intergalactic network?
>
> The hint was not lost on teve. uperBook? he asked slowly.
>
> Cranos let out a malicious grin. Behind him, the robots were busy wiring into S.H.I.M.'s core network. "While your SuperBook users will be busy playing SuperVille, the tentacles of Sentinel will spread into new unsuspecting worlds. Critical systems of every intelligent species will be sabotaged. The Supers will have to bow to a new intergalactic supervillain—Cranos." As Cranos was delivering this extended monologue, Steve noticed a movement in the corner of his eyes. It was Acorn, the super-intelligent squirrel, scurrying along the right edge of the rooftop. He also spotted Hexa hovering strategically on the other side. He nodded at them.
>
> Hexa levitated a garbage bin and ung it towards the robots. corn distracted them with high-pitched whistles. "Kill them all!" Cranos said irritably. s he turned to watch his intruders, teve fished out his phone, dialed into FaceTime and held it towards Cranos.
>
> "Say hello to your old friend, Cranos," said Steve. Cranos turned to face the phone and the screen revealed Madam O's face. With a smile, she muttered under her breath, "Taradiddle Bumfuzzle!" The expression on Cranos' face changed instantly. The seething anger disappeared. He now looked like a man they had once known.

>What happened? asked Hart confused.

> "We thought we had lost you," said Madam O over the phone. "I had to use hypnotic trigger words to bring you back." Hart took a moment to survey the scene around him. Then, he slowly smiled and nodded at her.
>
> `One Year Later`
>
> Who would have guessed Acorn would turn into an intergalactic singing sensation in less than a year? His latest albu corn Unplugged" debuted at the top of Billboard's Top 20 chart. He had thrown a grand party in his new white mansion overlooking a lake. The guest list included superheroes, pop stars, actors, and celebrities of all sorts.
>
> "So, there was a singer in you after all," said Captain Obvious holding a martini.
>
> "I guess there was," replied Acorn. He looked dazzling in a golden tuxedo with all sorts of bling-bling.

>teve appeared with Hexa in towwho looked ravishing in a owing silver gown. "Hey Steve, Hexa.... It has been a while. Is SuperBook still keeping you late at work, teve?

> "Not so much these days. Knock on wood," replied Hexa with a smile.

>h, you guys did a fantastic job. I owe a lot to uperBook. My first single, 'Warning: Contains Nuts', was a huge hit in the Tucana galaxy. They watched the video on SuperBook more than a billion times!"

> "I am sure every other superhero has a good thing to say about SuperBook too. Take Blitz. His AskMeAnything interview won back the hearts of his fans. They were thinking that he was on experimental drugs all this time. It was only when he revealed that his father was Hurricane that his powers made sense." By the way, how is Hart doing these days?
>
> "Much better," said Steve. "He got professional help. The sentinels were handed back to S.H.I.M. They are developing a new quantum cryptographic algorithm that will be much more secure."
>
> "So, I guess we are safe until the next supervillain shows up," said Captain Obvious hesitantly.
>
> "Hey, at least the beacon works," said Steve, and the crowd burst into laughter.

总结

In this final chapter, we looked at various approaches to ake your Django application stable, reliable, and fast. In other words, to make it production-ready. While system administration might be an entire discipline in itself, a fair knowledge of the web stack is

essential. We explored several hosting options, including PaaS and VPS.

We also looked at several automated deployment tools and a typical deployment scenario. Finally, we covered several techniques to improve frontend and backend performance.

The ost iportant ilestone of a website is finishing and taking it to production. However, it is by no means the end of your development journey. There will be new features, alterations, and rewrites.

very tie you revisit the code, use the opportunity to take a step back and find a cleaner design, identify a hidden pattern, or think of a better implementation. Other developers, or sometimes your future self, will thank you for it.

I'm digging a hole for you guys.